A Modal Calculus for Effect Handling

Aleksandar Nanevski

June 30, 2003

CMU-CS-03-149

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In their purest formulation, monads are used in functional programming for two purposes: (1) to hygienically propagate effects, and (2) to globalize the effect scope – once an effect occurs, the purity of the surrounding computation cannot be restored. As a consequence, monadic typing does not provide very naturally for the practically important ability to handle effects, and there is a number of previous works directed toward remedying this deficiency. It is mostly based on extending the monadic framework with further extra-logical constructs to support handling.

In this paper we adopt a different approach, founded on the observation of Pfenning and Davies that an abstract monad can be decomposed in terms of modal operators for possibility $\Diamond$ and necessity $\Box$. Our idea is to use the $\Box$ modality (which is a comonad) for hygienic propagation of effects, and leave the globalization of effect scope to $\Diamond$. Then the effects which admit a natural notion of handling can be encoded using $\Box$; since they are not global, there is no need to push them under $\Diamond$.

Based on this idea, we develop a general framework for effect handling systems, and obtain novel calculi for exceptions, catch-and-throw and composable continuations as specific instantiations.

Keywords: modal logic, effect systems, exceptions, composable continuations

1 Introduction

It is a well-established correspondence in functional programming that monads are a type-theoretic equivalent to effects and effect systems [38]. A monad, as described by Moggi [19, 20] and Wadler [35, 37], serves two purposes: (1) it marks the impure code segments of a program, supporting in this way a disciplined propagation of effects, and (2) because there is no canonical map to leave the scope of a monad, effects represented by monads become *global* (i.e., once introduced, the effect "holds" till the end of the program).

Therefore, neither monads nor the effect systems, in their purest formulation, can express the important, and in programming practice ubiquitous ability of *handling effects*; that is, restoring purity to an impure computation by means of some action. In fact, this very definition of handling effects is in direct contradiction to the above property (2) that effects should have global scope. The expressiveness of effect handling, when required, has to be endowed upon the monadic framework by means of additional extra-logical constructs [7, 8, 9, 15, 26].

In this paper we present a novel effect system, based on modal logic, which is capable of representing effect handling. It is founded on the observation of Pfenning and Davies in [28] that an abstract monad can be deconstructed as a composition $\Diamond\Box$ of modalities for possibility $\Diamond$ and necessity $\Box$ of a variant of modal logic S4. We build on this result by recognizing that, informally: (1) $\Box$ takes the duty of marking impure code segments and enforcing effect propagation discipline, and (2) $\Diamond$ takes the duty of single-threading the program and globalizing the scope of effects.

We can use the two independent modal type constructors to ascribe typings that are more precise than those of a monadic calculus. Those effects which admit a natural notion of handling may be encoded using only the $\Box$ modality; since the scope of such effects is not global, they need not be forced under a $\Diamond$.

The necessitation operator $\Box$ is in fact a *comonad*, and recently several authors have made an argument that comonads too, in addition to monads, can be used to represent certain kinds of effects. For example, [16] argues that comonads are more appropriate than monads to represent effects which arise from the environment. Also, [25] identifies comonads as encoding computations which expect additional arguments from the environment.

In this paper, we focus on the necessitation operator $\Box$, and leave the issues related to global effects (e.g. destructive state update) and the possibility operator $\Diamond$, for a related paper [24]. With this development, we hope to further strengthen the case for comonads. The idea is to extend the modal $\lambda^{\Box}$-calculus from [28] with a new semantic category of names [23] that would be used to label and track specific effects. Thus, similar to indexed monads in [38], we will have a type $\Box_C A$ classifying *suspended* computations of type A which may, in the course of eventual execution, raise effects listed in the (possibly ordered) set of names C. Unlike with monads, canonical coercions from $\Box_C A$ to A are possible, and they represent *handlers* which are equipped to deal with names in C.

In this sense, the comonad of modal necessity, when endowed with names, not only represents effects arising from the environment, but in fact marks effects whose scope is not global – effects which can be handled. Also, the interaction of the environment with the comonad and the aforementioned passing of additional arguments, can be very diverse – different effects will have different notions of handling.

This approach in formulating effects follows the introduction/elimination pattern of natural deduction; effects are introduced by their introduction forms, and are eliminated by their handling forms. This is particularly appealing because it may uncover the logical content of the effects in question (although we do not explore this further). To illustrate the idea we develop as an example a novel calculus of exceptions [5, 26], in which each exception is associated with a name, raising exceptions is the introduction form and handling exceptions is the elimination form. We also present novel calculi for catch-and-throw [22, 14] and composable continuations [6, 2, 3, 21, 36, 11, 12, 13]. All these calculi are derived as simple and uniform extensions of our comonadic core framework.

2 Modal $\lambda^{\Box}$-calculus

The starting point for the development of our language of effects is the $\lambda^{\Box}$-calculus of [28, 4]. The $\lambda^{\Box}$ is the proof-term system for the necessitation fragment of modal logic S4, and it was first considered in functional programming in the context of specialization for purposes of run-time code generation [4, 39, 40]. 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	$::=$	$x \mid u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid$ $\textbf{box}\ e \mid \textbf{let box}\ u = e_1\ \textbf{in}\ e_2$
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 S4 modal logic it is a necessitation modifier on propositions [28]. For the purposes of this paper, a useful operational intuition is to consider the type $\Box A$ as a type of *suspended expressions of type* A. In contrast, the non-modal type A would be populated with only *executable expressions*. In this sense, rather than suspending expressions by enclosing them under a λ-binder, as it is customarily done in functional programming, we use a separate language construct for that. This gives us an orthogonal and more appropriate abstraction mechanism that we will build on in subsequent sections.

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 executable expressions, and Δ for variables bound to suspended expressions. 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. Note that the typing rule for **box** prohibits e to refer to variables from Γ; it is not possible to coerce values into suspensions. This is counter-intuitive to our interpretation of the modal calculus and we will remedy it in subsequent sections. The elimination form **let box** $u = e_1$ **in** e_2 takes the suspended expression boxed by e_1 and binds it to the expression variable u to be used in e_2.

Example 1 The function `exp2` below takes an integer argument n and builds a suspension for computing 2^n.

```
fun exp2 (n : int) : □int =
    if n = 0 then box 1
    else
      let box u = exp (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 suspended positions (i.e., under a box) and executable positions. This way we can compose suspended programs, 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
```

To formalize the above operational intuition about the calculus, we employ an evaluation context operational semantics in the style of Wright and Felleisen [41]. We have decided on a call-by-value evaluation strategy which, in line with our interpretation of boxed expressions as suspended code, prohibits reductions under box; boxed expressions are considered values. This choice is by no means canonical, but is necessary for the purposes of this paper. The formalization relies on the definitions of redex and evaluation context introduced below.

$$\begin{array}{lcll}
\textit{Values} & v & ::= & x \mid \lambda x{:}A.\ e \mid \mathbf{box}\ e \\
\textit{Redexes} & r & ::= & v_1\ v_2 \mid \mathbf{let\ box}\ u = v\ \mathbf{in}\ e \\
\textit{Evaluation contexts} & E & ::= & [\ ] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e
\end{array}$$

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 (which we denote by $\longmapsto$) always first reduce the redex identified by the unique decomposition. The primitive reduction and the evaluation relation for call-by-value $\lambda^{\Box}$ are defined as follows.

$$\begin{array}{rcl}
(\lambda x{:}A.\ e)\ v & \longrightarrow & [v/x]e \\
\mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 & \longrightarrow & [e_1/u]e_2
\end{array}$$

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

3 Names as markers for effects

In this section, we extend the $\lambda^{\Box}$-calculus 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 suspended term, then the corresponding $\Box$-type will be indexed by that name. For example, if we have an exception X, then a suspended term of type A which may raise this exception, will be given a type $\Box_X A$, and we also provide coercions from $\Box_X A$ to A which would represent exception handlers for X.

The described indexing of the modal operator with names is similar to the one found in the monadic language from [38], 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. How names relate to 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.

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

Just like $\lambda^\Box$, our calculus makes a distinction between levels of suspended and executable expressions. 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 suspended expressions which 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 suspended expressions 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. Furthermore, as already commented in the previous section, we would like to enable coercion of values into suspended expressions, so we join the two contexts of the $\lambda^\Box$ calculus into one. The context Γ is now considered part of Δ declaring variables with explicitly empty support. Correspondingly, we will frequently abbreviate $x{:}A[\ ]$ as $x{:}A$. This decision logically corresponds to identifying the types A and $\Box A$, where the index support in the later type is empty. Equivalently, it can be viewed as imposing monotonicity on the Kripke frame in the possible worlds semantics for the S4 modal logic [28]. Of course, because now we have a multitude of modal operators corresponding to various supports that can be used as indices, this move does not collapse the whole hierarchy of modal types.

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** picks 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 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 explained 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

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

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

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

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

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

Figure 1: Type system of the core fragment.

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 suspended under a **box** (and correspondingly accounted for in the type of e). This parallels exactly the 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 in 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 suspensions. 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. A formulation of the type system which does not favor any particular evaluation strategy is possible; it retains the two contexts from Section 2, and achieves the discussed monotonicity by means of a separate rule. Therefore, it is a bit more verbose, so we leave it for future work.

Modal fragment. To type a suspended code **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, 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.

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 appears in some dead-code part of e), or all the occurrences of X are handled by an appropriate notion of 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 2 We can introduce the construct **let val** $x = e_1$ **in** e_2 into the calculus, using the rule

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

This is syntactic sugar for **let box** $u = (\lambda x.\ \textbf{box}\ e_2)\ e_1$ **in** u, rather than the usual $(\lambda x.\ e_2)\ e_1$. The complication arises because we have to box e_2 and make it pure, before we can put it under a λ-abstraction.

Example 3 Assume that C_1, C_2 and D are arbitrary support sets. Then the following terms are well-typed of *empty support*.

$$\begin{aligned}
f_1 &: \Box_{C_1} A \to \Box_{C_2} A = \qquad \text{(where } C_1 \sqsubseteq C_2\text{)}\\
&\quad \lambda x.\ \textbf{let box}\ u = x\ \textbf{in box}\ u\\
f_2 &: \Box A \to A =\\
&\quad \lambda x.\ \textbf{let box}\ u = x\ \textbf{in}\ u\\
f_3 &: A \to \Box_D A =\\
&\quad \lambda x.\ \textbf{box}\ x\\
f_4 &: \Box_{C_1} A \to \Box_D \Box_{C_2} A = \qquad \text{(where } C_1 \sqsubseteq C_2\text{)}\\
&\quad \lambda x.\ \textbf{let box}\ u = x\ \textbf{in box}\ (\textbf{box}\ u)\\
f_5 &: \Box_{C_1}(A \to B) \to \Box_{C_2} A \to \Box_D B = \qquad \text{(where } C_1, C_2 \sqsubseteq D\text{)}\\
&\quad \lambda x.\ \lambda y.\ \textbf{let box}\ u = x\ \textbf{in let box}\ v = y\ \textbf{in box}\ (u\ v)
\end{aligned}$$

The function f_1 simply eta-expands its argument. It shows that support weakening in boxed types is derivable. The function f_2 illustrates that we can "unbox" and evaluate suspended expressions which *do not raise any effects*; notice that the argument type of f_2 has empty index support. The function f_3 shows that, as a consequence of the imposed monotonicity, it is possible to coerce values into suspended computation, by simply boxing them. Because values are name-free, their support can be weakened to an arbitrary support set D. The other two functions generalize the characteristic comonadic axioms of S4 modal necessity without names [4, 28].

Example 4 Anticipating Section 6, 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 suspension $(\mathbf{box\ raise}_Y\ e_2){:}\Box_{X,Y} A$. 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 suspension 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 6 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 suspended, 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 suspended, never raised or otherwise handled. The definition of the judgment relies on the notion of redexes 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	$::=$	$[\,] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid \mathbf{choose}\ E$

The primitive reduction rules and the evaluation rules of the operational semantics are presented below. They are identical to the rules for $\lambda^{\Box}$, except for the new reduction rule for **choose** $(\nu X{:}A.\ e)$, which extends the run-time name context Σ with a fresh name X before proceeding with the evaluation of e.

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

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

Example 5 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 4.

$(X{:}A), \textbf{choose}\ (\nu Y{:}A.\ (\lambda x{:}\Box_{X,Y} A.\ e_1)\ (\textbf{box raise}_Y\ e_2)) \longmapsto$
$(X{:}A, Z{:}A), (\lambda x{:}\Box_{X,Z} A.\ e_1)\ (\textbf{box raise}_Z\ e_2) \longmapsto$
where Z is a fresh name
$(X{:}A, Z{:}A), e_1 \longmapsto$
$\ldots$

5 Structural properties and type soundness

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* $v = \textbf{box}\ e$ *and* $\Sigma; \cdot \vdash e : B\,[D]$
3. *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 (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 $e_2 = \pi(e_1)$.*

6 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.

$$\begin{array}{llll} \textit{Exception handlers} & \Theta & ::= & \cdot \mid X z \to e, \Theta \\ \textit{Terms} & e & ::= & \ldots \mid \mathbf{raise}_X\ e \mid e\ \mathbf{handle}\ \langle\Theta\rangle \end{array}$$

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 Θ is syntactically defined as a list of exception patterns, each of which refers to a different exception. More conceptually, we can regard the exception handler as a mapping from the set of names to the set of functions over terms

$$\Theta : \mathcal{N} \to \textit{Terms} \to \textit{Terms}$$

We will treat our handlers as being able to take action on *all* the exceptions; it is just that on some of them the action will be a specifically prescribed term, while on others the action will just involve propagation of the exception. For example, the empty handler $\langle\;\rangle$ handles all the exceptions by propagating them further.

Given a handler Θ, its domain $\mathbf{dom}(\Theta)$ is defined as the set

$$\mathbf{dom}(\Theta) = \{X \in \mathcal{N} \mid \Theta(X)(z) \neq \mathbf{raise}_X\ z\}$$

We only consider handlers with *finite* domains. A handler Θ with a finite domain has a finitary syntactical representation as a set of patterns $Xz \to e$ relating a name X from $\mathbf{dom}(\Theta)$ with the function $\Theta(X) : z \mapsto e$. We will frequently equate a handler and the set that represents it when it does not result in ambiguities. The operations that e can perform on the term z in the above mapping are limited to only ordinary compositions with the term constructors from our language. Thus, it will be the case that the value of the function $\Theta(X)$ at a term e' is equal to $[e'/z]e$.

Example 6 Assuming X and Y are integer names, the following are well-formed expressions of the exception calculus.

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

The terms evaluate to 13, 0, and 1, respectively. The first term raises the exception Y and then handles it. The second raises X with value 0, which is then handled by the first handler, but this handler itself raises the exception Y with value 0, ultimately handled by the second handler. The third term raises X with value 0, which is propagated by the first handler, and then handled by the second handler.

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 in these supports 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

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

and the handler Θ will be given the type $A[C] \Rightarrow A[D]$ 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

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

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

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

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

Figure 2: Typing rules for exceptions.

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 7 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 $\Box_{\texttt{EMPTY}}$`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. To reduce clutter, in this and in subsequent examples we abbreviate **let box** $u = (-)$ **in** u simply as **unbox** u.

```
- 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;
```

There are several points worth emphasizing about our calculus. First of all, exceptions in our calculus are not values and cannot be bound to variables. Correspondingly, they 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. This is in contrast to the general mechanism of monads [37] where effects are global, and therefore adding an effect to the body of a pure function invariably requires that the range type of the function be changed to a monadic type. This in turn may require serious restructuring of the programs that use such a function.

Finally, our calculus presents really only a bare-bone theoretical foundation for the treatment of exceptions. As is probably the case with other effect calculi too, it is not very practical when compared to, say, Standard ML, exactly because the types are decorated with exception names. But, we believe that such a hurdle can be overcome; it is easier to hide the excess information, than to recover from the lack thereof, and we comment on this in the future work section.

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 $E = E'[P'\ \mathbf{handle}\ \langle\Theta\rangle]$.

The primitive reduction on the new redexes follows.

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

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 6, 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* $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : B[D'] \Rightarrow B[D]$*, then* $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : B[D'] \Rightarrow B[D]$

Lemma 11 (Unique decomposition) *For every expression* e*, either:*

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

Theorem 12 *The calculus satisfies progress and preservation.*

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

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

Figure 3: Typing rules for catch and throw.

7 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, however, in order to illustrate a different notion of handling. It will also provide some intuition for the calculus of composable continuation in Section 8. 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. The exact syntax of the two constructs is defined as follows.

$$\mathit{Terms} \quad e \quad ::= \quad \ldots \mid \mathbf{throw}_X\ e \mid \mathbf{catch}_X\ e$$

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 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 8 The following terms (adopted from [12]) are well-typed in our catch-and-throw calculus.

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

```
choose (νX:int.
  (λf:int->□_X int.
               let box u = f (box 0)
               in
                   1 + catch_X u
               end) (λy:int. box (throw_X 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 9 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) : □_EXIT intlist =
           case xs
            of nil => box 1
           | x::xs =>
                if x = 0 then box (throw_EXIT 0)
                else
                  let box u = mult xs in box(x * u)
   in
       catch_EXIT (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.

Redexes	r	$::=$	$\ldots \mid \mathbf{catch}_X\ v \mid \mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$
Evaluation contexts	E	$::=$	$\ldots \mid \mathbf{catch}_X\ E \mid \mathbf{throw}_X\ E$

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

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

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 13 (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.*

Theorem 14 *The calculus satisfies progress and preservation.*

8 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 **reset** which also assigns a name to it; thus, it is similar to **catch** from the previous section. The jump is itself is performed by **shift**, which in a sense corresponds to **throw** from the catch-and-throw calculus. The exact syntax of the calculus is defined as follows.

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

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 $\mathbf{shift}_X\ k.\ e$ is evaluated, it captures into the variable k the part of the surrounding term between this **shift** and corresponding **reset** 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 shift to destination points which were defined in the captured and removed part.

Example 10 This example is adopted from [2, 36]. The terms below are well-typed examples in our calculus of composable continuations.

```
e₁ =  1 + reset_X (10 + shift_X f:□_X int->□_X int.
                let box u = f (f (box 100))
                in
                    reset_X u
                end) ⟼* 121
e₂ = 1 + reset_X (10 + shift_X f. 100) ⟼* 101
e₃ = 1 + reset_X (10 + shift_X f.
               let box u1 = f (box 100)
                   box u2 = f (box 1000)
               in
                   reset_X (u1 + u2)
               end) ⟼* 1121
```

In each of these examples, the continuation variable $f{:}\Box_X int \to \Box_X int$ is bound to $\lambda x.$ **let box** $v = x$ **in box** $(10 + v)$, which represents the environment delimited by the **shift** and the corresponding **reset**. Notice that upon capturing of the continuation, the delimiting **reset** 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 our calculus for composable continuations is again $\Sigma; \Delta \vdash e : A\,[C]$. It establishes that the expression e has type A and may shift to 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 the shifting to a name that is the notion of effect, and establishing a destination point is the notion of handling. Therefore, the type system should enable a **shift** to a destination point X only if X is accounted at the support C, placed there by a corresponding **reset**. The situation, however, is a bit more involved. As already mentioned, $\mathbf{shift}_X\ k.\ e$ evaluates e in a changed environment from which the destination points between X and the **shift** have been removed; thus e has to be typechecked against a support that is changed correspondingly.

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 effect calculi where supports were simply sets, here we actually must exploit their list-like structure. To simplify matters, we allow a **shift** to a certain name only if that name is at the end of the support. This is accounted for in the typing rule for $\mathbf{shift}_X\ k.\ e$ which demands that X is the *rightmost* name in the support (C, X). After the environment delimited by X has been captured and removed, the destination point X is removed as well, and e is evaluated in the changed environment. Correspondingly, e is typechecked against support from which the rightmost X has been removed. If a shift is required to a name which is deeper down in C, it can still be done by performing a sequence of nested shifts in a last-in-first-out manner to all the names above. In that sense, we can actually view the supports of our calculus of continuations as *stacks*.

There are yet further important aspects of the typing rule for **shift** 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 **shift** does not matter; we can type **shift** by an arbitrary type B. The domain and the range of the continuation k must match the source and the destination points of the **shift**, which in this rule have types B and A, respectively. The **shift** appears in the context of a support stack (C, X) and that is why k is placed into the context with the domain type $\Box_{C,X}B$. The range type of k is $\Box_{C,X}A$, meaning that the captured continuation *will not include* the delimiting $\mathbf{reset}_X$ (otherwise, this $\mathbf{reset}_X$ would have been a handler for X, making the range type $\Box_C A$). This is in contrast to most other calculi for composable continuations [6, 2, 3, 21, 36, 11, 12, 13] which either capture the delimiting **reset** into the continuation, or leave it in the environment, or both. Our decision to do neither, actually adds further expressiveness to the calculus, and we illustrate this point at the end of the section.

The typing rule for **reset** is much simpler. The construct $\mathbf{reset}_X\ e$ establishes a destination point X and allows the expression e to shift to X by placing X into 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 11 The program below is a particularly convoluted way of reversing a list, adopted from [2].

```
fun reverse (l : intlist) : intlist =
 choose (νX: intlist.
  let fun rev' (l : intlist) : □X intlist =
      case l
        of nil => box nil
      | (x::xs) =>
          let val y = rev' xs
          in
              box (shiftX c:□X intlint -> □X intlist.
                     resetX x :: unbox (c y))
          end
      box v = rev' l
  in
     resetX v
  end)
```

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

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

Figure 4: Typing rules for composable continuations.

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

```
box (shiftX c3.
      resetX 2 :: unbox c3 (box shiftX c2.
        resetX 1 :: unbox c2 (box shiftX c1.
          resetX 0 :: unbox c1 (box nil)))
```

When prepended by a $\texttt{reset}_X$, 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.

As in the development of operational semantics for the previous effect calculi, here too we start by extending the notion of evaluation contexts from the core language, and defining the new redexes.

Evaluation contexts	E	$::=$	$\ldots \mid \mathbf{reset}_X\ E$
Pure contexts	P	$::=$	$[\,] \mid P\ e_1 \mid v_1\ P \mid \mathbf{let\ box}\ u = P\ \mathbf{in}\ e \mid \mathbf{choose}\ P$
Redexes	r	$::=$	$\ldots \mid \mathbf{reset}_X\ v \mid \mathbf{reset}_X\ P[\mathbf{shift}_Y\ k.\ e]$

Because each **shift** is handled by the nearest **reset** (and the typing rules ensure that these are labeled by the same name), we need to identify within each evaluation context E that **reset** (if any) which is closest to the hole of E. Thus, similar to the calculus of exceptions, we identify the specific subclass of evaluation contexts which are pure, in the sense that they do not contain any resets. Then, it is easy to prove that each evaluation context E is either pure, or there exist unique evaluation context E' and pure context P' such that $E = E'[\mathbf{reset}_X\ P']$. The primitive reduction rules for the new redexes are defined below.

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

The first reduction rule is simple; it just serves to let the values pass through a **reset**. Indeed, values are effect free (in this case, shift-free), so no resets are really relevant for them. The second primitive reduction deserves more comment. It prescribes that the evaluation context P be captured and substituted for a continuation variable k in e. But notice that the reset which delimits the context P is *discarded* altogether from the further evaluation. It does not survive the capturing as part of the reduct, and it does not survive in the captured continuation either (which only includes P). We have already pointed this fact out in our discussion of the type system, and this reduction rule makes it formal.

There are merits to both of these choices. The more obvious one is that because the **reset** is removed from the environment, we can shift to names which are further down in the support stack. It is a bit more difficult to explain the benefits of the decision that the delimiting **reset** be discarded from the captured continuation as well. From a purely technical aspect, it certainly seems more flexible to discard the **reset** than to include it in the continuation. After all, if the **reset** is missing, we can always put it back; if it is retained, we can never eliminate it. But the important point is that it also improves the expressiveness of the calculus. A particular example we have in mind concerns the application of composable continuations to neatly encode bounded nondeterministic computation [2, 3] where *depth-first search* is employed to explore the space of solutions. What we would be able to do in our system, thanks to this particular design decision, is to express in the same manner the *breadth-first search* as well. We illustrate this argument in the following two examples.

Example 12 [Depth-first search] Composable continuations have been used to conveniently express "nondeterministic computation"; that is, computation which can return many results [2, 3]. We paraphrase from these papers the following program for finding all the triples (i, j, k) of distinct positive integers smaller than n that sum up to s, which is very effectively phrased in terms of a primitive function `choice`.

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

```
(* triple : int*int->unit *)
fun triple (n, s) =
  reset_X
    let val i =
           unbox (choice n)
        val j =
           unbox (choice (i-1))
        val k =
           unbox (choice (j-1))
    in
        if (i+j+k = s) then
          print (i, j, k)
        else ()
    end
```

When run with $n = 9$ and $s = 15$, the function `triple` should print out the triples $(9,5,1)$, $(9,4,2)$, $(8,6,1)$, $(8,5,2)$, $(8,4,3)$, $(7,6,2)$, $(7,5,3)$ and $(6,5,4)$ before returning $() : \mathit{unit}$. The program works by maintaining a list of composable continuations, which is represented at run-time as a sequentially composed sequence of expressions of unit type. Each call to `choice` picks the *top element* of the list and stores this top element into the composable continuation `c`, delimited by a name $X : \mathit{unit}$, while at the same time removing it from the list. This element is then expanded into: `(reset`$_X$ `c n; reset`$_X$ `c (n-1);...;reset`$_X$ `c 1; reset`$_X$ `())` which is placed back at the top of the list. In this sense, the strategy that the above program uses for exploring the search space is *depth-first*. The depth of the search tree must be *bounded* (and it is in this particular example) if the program is to *enumerate* all the solutions. This is the relation of composable continuations in the style of [2, 3] to bounded non-determinism.

Example 13 [Breadth-first search] In our calculus of composable continuations we are not limited to the depth-first-search strategy. We can, for example, employ *breadth-first-search* to enumerate the solutions even if the search space is of unbounded depth. As outlined before, our continuations capture the environment up to, but not including the resets, and this provides the needed expressiveness. In particular, when expanding the top element from the list of computations, we do not need to push the newly expanded cases on the top of the list (as in the previous example); we can place them at the bottom. How does this work? First, we capture the whole list of computations into a continuation function of its own. The argument of this function abstracts the top of the list, and therefore, adding new elements at the top of the list represented by the continuation (i.e. doing depth-first-search) is not influenced by whether the continuation is delimited by a **reset** of not. But, the continuations must not be delimited if elements are to be added at the bottom, as otherwise the delimiter gets in the way. Because we do not capture the delimiting **reset** into the continuation, we can first extend the represented list at the bottom (thus encoding breadth-first-search strategy), and only then place a **reset** back, to delimit the new list. The scope of this **reset** will then include the newly expanded search cases.

To paraphrase, even though our continuations are not delimited at the time of capturing, the type system will force us to delimit them before they are invoked, but not before we had a chance to modify them first to suit our needs.

As an example, we present a version of `triple` that uses breadth-first-search strategy. We now need two names to delimit the continuations: the name $X{:}\mathit{unit}$ which delimits individual entries in the list, and the name $Y{:}\mathit{unit}$ for the whole list. The new function `bfchoice` picks the top element `c1` expands it into: `(reset`$_X$ `c1(n); reset`$_X$ `c1(n-1);...; reset`$_X$ `c1(1); reset`$_X$ `())`. Then this whole expanded segment is attached to the bottom of the list, which itself has been captured into the continuation `c2`.

```
(* bfchoice:int->□_{Y,X}int *)
fun bfchoice n =
  box (shift_X
        c1:□_{Y,X}int->□_{Y,X}unit.
        shift_Y
         c2:□_Y unit->□_Y unit.
    reset_Y
     let fun loop (s:int):unit =
         if s = 0 then ()
         else
           let box u = c1(box s)
           in
             (reset_X u);
             loop (s - 1)
           end
         box v =
           c2(box reset_X ())
     in
       v; loop (n)
     end)

(* triple:int*int->unit *)
fun triple (n, s) =
  reset_Y
   reset_X
    let val i =
           unbox (bfchoice n)
        val j =
           unbox (bfchoice(i-1))
        val k =
           unbox (bfchoice(j-1))
    in
        if (i+j+k = s) then
          print (i, j, k)
        else ()
    end
```

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

Lemma 15 (Unique decomposition) *For every expression e, either:*

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

Theorem 16 *The calculus satisfies progress and preservation.*

9 Related work

Integrating effects into functional calculi has quite a long history, and this section is bound to be very incomplete. Numerous systems have been proposed, treating various effects and with various levels of precision and verbosity of typing. As an example, we only list [17, 32, 33, 34].

A treatment of exceptions in Haskell is given by Peyton Jones et al. in [26]. This paper associates exceptions with values, so that a value of any type is either "normal" or "exceptional". This is similar with our calculus where expressions, rather than function calls, are marked by the type system as exceptional. The paper further introduces handling of exceptions by means of a primitive map `getException` which must be enclosed within the IO monad, to preserve soundness. Another exception calculi is presented by de Groote in [5]. It is a call-by-value calculus which uses separate binding mechanisms to introduce exceptions into the computation. The calculus lacks modal or monadic types, so it has to specifically require that values of the language are effect-free; in this case it implements the Standard ML exception mechanism. This paper also discuses the logical content of exceptions, and relationship with classical logic.

Exception mechanism of Java relates to our calculus as well, as the Java methods must be labeled by the exceptions they can raise [10]. Catch-and-throw calculus is a specific simplifications of exceptions, and theoretical analysis of catch and throw can be found in [22, 12, 14].

Composable continuations were probably first considered by Felleisen in [6], in an untyped setting and with shifting only to the nearest reset (or prompt). Sitaram and Felleisen in [31] generalized this to a whole family of control operators for shifting, each of which is indexed by a numeral prescribing how many closest resets should be jumped over. Also in untyped setting, Hieb, Dybvig and Anderson in [11] introduce labels instead of numerals to describe the destination points for a hierarchy of shifts. Danvy and Filinski in [2] develop a type system for composable continuations with a single shift operator. The resets are not labeled. In the Appendix C, they also briefly discuss the idea which we have employed here: upon capturing, remove the resets from the environment, so that jumps can be made to the resets further down in the context stack. Logical content of composable continuations is studied by Murthy in [21]. This paper develops a type system for composable continuation with a hierarchy of shifting operators, which is based on monads indexed by lists of types, but has to restrict the resets to only implication-free types in order to preserve soundness. Wadler in [36] further analyses the above type systems for composable continuations with a single shift operator, and with a hierarchy of shift operators, and presents them in terms of indexed monads. Most recently, Kameyama in [12, 13] works with labels instead of numerals to provide a hierarchy of shifting operators. These calculi lack modal or monadic types and must (like the above-mentioned system of exceptions [5]) limit the calculus in order to avoid scope extrusion for labels and preserve soundness.

Coming from the side of logic and type theory, effect calculi are directly representable by monads. Monads were first used in denotational semantics by Moggi [19, 20], and were adopted for functional programming by Wadler [35, 37, 38] to represent effectful computations. We further point to the work of Filinski [7, 8, 9] which develops the concepts of monadic reflection and reification, with intentions similar to ours: to increase the flexibility of monadic programming and delimit the scope of effects. Similar motivation lies behind Kieburtz's introduction of lexical scoping for effects in [15].

On the other hand, our paper was specifically motivated by the work of Pfenning and Davies [28] which formulates natural deduction for a variant of S4 modal logic and a proof-term calculus for it. It also shows how a monadic system can be embedded using modalities. The modal necessity, as defined in this paper, is a comonad, and the fact that comonads may represent effects have already been noticed by Kieburtz [16] and Pardo [25], but the relation to handling was not established.

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 on Nominal Logic and FreshML by Pitts and Gabbay [30, 29] which introduce names as urelements of Fraenkel-Mostowski set theory.

10 Conclusions and future work

This paper introduced a logically motivated effect system, based on modal logic S4, which is designed to naturally and *uniformly* support the process of effect handling. It is founded on the observation by Pfenning and Davies in [28] that a monad can be decomposed in terms of operators for necessity and possibility of a variant of modal logic S4. In our calculus, the necessity operator $\Box$ serves to mark impure program segments and accounts for a hygienic propagation of effects; it is used to represent effects which can be handled. We also give a call-by-value operational semantics for the language, but a purely logical formulation which does not favor any particular evaluation strategy, should be possible; we hope to explore such a system and its equational theory in a future work.

The presented system extends the modal calculus from [28] with names, which are labels that can be dynamically introduced into the computation and serve to identify effects. The (possibly ordered) collection of effects that may appear in a certain term is refered to as support of the term. The typing judgment relates terms to their supports, and only terms of empty support (which are, hence, guaranteed to be effect-free) will be considered for evaluation. The notion of support in the calculus is rather flexible; depending on the particular effect being modeled, different definitions of support may be used. For example, in case of exceptions and catch-and-throw, the supports are simply sets of names listing the exceptions and jump destinations, respectively. In case of composable continuations, supports are stacks of names depicting the environments of nested continuation-delimiting points. It is an interesting future work to investigate how different definitions of support may interact in a system combining several effects that are heterogeneous in this respect.

In our system, the modalities are indexed by supports. For example, the type $\Box_C A$ classifies computations of type A capable of raising effects whose names are listed in the support C. It is possible to project a value of type A out of a computation of type $\Box_C A$ if the effects in C can be handled (in a way specified by the particular effects in question). In the special case when C is the empty support, the computation is effect-free, and the value can be projected out trivially; this is the behavior of the propositional S4 necessity, recovered. We further impose monotonicity on the new logic with multiple modalities, resulting in the identification of the types A and $\Box A$, where the later type has empty index support. Operationally, this allow us to coerce values into effectful computations, much like a monadic system would do.

The presented system consists of a core effect calculus which can easily be extended to account for various effects, simply by providing the introduction and elimination (i.e., handling) forms and specifying the particular notions of support suitable for that effect. We demonstrate this by extending the core fragment into new calculi for exceptions, catch-and-throw and composable continuations. We should mention, however, that the obtained systems are probably not very practical as the verbosity of support annotations on types may significantly complicate any serious programming effort. It may be of some comfort that a similar criticism is in fact applicable to most other effect calculi as well.

At any rate, it is a very important future work to investigate the questions of type and support inference in this calculus (possibly in the style of [32]), and develop new abstraction mechanism which can hide the unwanted support and result in a more practical language (perhaps combining support polymorphism [23] and proof irrelevance [27, 1]). The system presented here is logically motivated and fully explicit about the supports of its terms, and is therefore a solid theoretical basis for such investigations.

Acknowledgments The author would like to thank Frank Pfenning for his numerous suggestions regarding this paper.

References

[1] S. Awodey and A. Bauer. Propositions as [Types]. Technical Report IML-R–34-00/01–SE, Institut Mittag-Leffler, The Royal Swedish Academy of Sciences, 2001.

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

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

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

[5] 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.

[6] 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.

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

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

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

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

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

[12] 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.

[13] 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.

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

[15] R. B. Kieburtz. Taming effects with monadic typing. In *International Conference on Functional Programming, ICFP'98*, pages 51–62, Baltimore, Maryland, 1998.

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

[17] 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.

[18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition 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] A. Nanevski. From dynamic binding to state via modal possibility. In *International Conference on Principles and Practice of Declarative Programming, PPDP'03*, pages ??–??, Uppsala, Sweden, 2003. To appear.

[25] A. Pardo. Towards merging recursion and comonads. In *Proceedings of the 2nd Workshop on Generic Programming, WGP'00*, pages 50–68, Ponte de Lima, Portugal, 2000.

[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. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Symposium on Logic in Computer Science, LICS'01*, pages 221–230, Boston, Massachusetts, 2001.

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

[29] 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.

[30] 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.

[31] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

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

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

[34] H. Thielecke. From control effects to typed continuation passing. In *Symposium on Principles of Programming Languages, POPL'03*, pages 139–149, New Orleans, Louisiana, 2003.

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

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

[37] 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.

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

[39] 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.

[40] 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.

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