A logical interpretation of Java-style exceptions

Jeffrey A. Vaughan

Harvard University

Abstract. This paper presents a novel, type-based exception analysis for EC (an Exception Calculus)—a higher-order, typed programming language with first-class, Java-style exceptions. EC exceptions are predeclared and are subject to a nominal subtyping relation. Every exception has its own type, and generic exception handling is performed by handling a supertype of all applicable exceptions.
Typing is based on *result contexts*, which are are analogous to the multiple-conclusion succedents in Gentzen's LK. This paper proves the syntactic soundness of EC, and presents a shallow embedding of EC types and judgments in LK. The embedding gives rise to a pleasant interpretation of subtyping as logical consequence.

1 Introduction

Most programming languages allow expressions to produce side effects, such as mutating the heap or raising exceptions. Such effects make programs hard for programmers to reason about and compilers to optimize. This problem may be mitigated by type-based static analysis. Well known examples include IO monads as in Haskell, type and effect systems [15], and checked exceptions as in Java.

Haskell-style monadic reasoning is inspired by category theory, but many other effects analyses are ad-hoc. One notable exception is a line of work started by Nakano [12, 13] that performs exception tracking using ideas from Gentzen's [3] system LK. Additionally, languages such as $\lambda\mu$ and the dual calculus provide precise characterizations of some control effects in exotic reduction systems.

This paper presents a novel, type-based exception analysis for EC (an Exception Calculus)—a higher-order, typed programming language with Java-style exceptions. As in Java, EC exceptions are predeclared and are subject to a nominal subtyping relation. Every exception has its own type, and generic exception handling is performed by handling a supertype of all applicable exceptions. Exceptions are first class and may be, for example, raised after being passed to a function.

This region of the design space differs from ML, which also features first class exceptions. ML exceptions can be created generatively at runtime [11], and ML programmers implement generic exception handlers using wildcard pattern matching, not subtyping. Also, all ML exceptions inhabit the single type `exn`.

EC's exception analysis synthesizes ideas from Nakano [13] and from type-and-effects-languages [15]. An expression is associated with both a return type

and the set of exceptions that it might raise. These are presented uniformly in a *result context*. Consider

$$e \equiv \mathsf{if}\ b\ \mathsf{then}\ 3\ \mathsf{else}\ (\mathsf{raise}\ \mathsf{Fail}\ \text{"oops"}).$$

The typing judgment will assign it (modulo subtyping) the following result context.

$$e : (\mathsf{ret} : \mathsf{int}, \mathsf{exn}\ \mathsf{Fail} : \mathsf{string})$$

Here we see that e has two *result channels*. It may return an int, or it may raise a string-carrying Fail exception. Abstracting over e internalizes the entire result context, not just the return channel.

$$\lambda b : \mathsf{bool}.\ e : (\mathsf{ret} : \mathsf{bool} \to (\mathsf{ret} : \mathsf{int}, \mathsf{exn}\ \mathsf{Fail} : \mathsf{string}))$$

This type captures e's exceptions as *latent effects* [6]. Intuitively, latent effects are suspended exceptions which may be raised during a function call; here they correspond to the throws component of a Java method signature. Section 2 details EC's semantics and theory.

Because EC's type system assigns multiple-channel result contexts to expressions, it resembles sequent systems with multiple conclusions. Gentzen's LK [3] is the prototypical example. LK judgments have the form $\Phi \vdash \Theta$ where Φ and Θ are lists of propositions. The judgment is intended to hold when the conjunction of the Φs implies the disjunction of the Θs.

The relation between EC and LK extends beyond syntactic similarity. EC typing derivations have form $e : \Sigma; \Gamma \vdash \Delta$ where Γ assigns types to variables, and Δ is a result context. (Σ defines the exception hierarchy.) Intuitively, such a derivation may be translated into an LK derivation ending with $[\![\Gamma]\!] \vdash [\![\Delta]\!]$. Furthermore, the EC subtyping derivations ending in $\Sigma \vdash \tau_1 <: \tau_2$ corresponds to LK derivations ending in $[\![\tau_1]\!] \vdash [\![\tau_2]\!]$. Section 3 provides an explicit account of the translation and its properties.

EC's design embraces a tension between type and effect reasoning—which is precise but specialized to call-by-value reduction—and an approach based on a more literal formulation of logical realizers for LK. We intend for EC to represent a practical compromise between these models. By contrast, other logic-based effects systems stay closer to LK. For example, Nakano [12] can only type $\lambda x.e$ when e has no unhandled exceptions. (The programmer must use source-level "tag abstraction" to express e's exception channels as part of its return type.) Another approach [13] types the entire abstraction using the same, unsuspended, effects as e. Such techniques appear to yield general analyses that are sound for full reduction. However, they lack EC's simplicity and precision for call-by-value evaluation.

Above all, the design of EC is motivated by a desire to make make rigorous the sort of effects analysis performed in a language like Java, and to show that this analysis has a clear, proofs-as-programs interpretation. Toward this end, we keep EC's feature-set minimal and ignore interesting but extraneous issues such as modules, separate compilation, and effects polymorphism.

This paper's main contributions follow.

– We define system EC, a language with a sequent-style effects system. It treats latent effects properly for call-by-value evaluation.
– We prove syntactic soundness of EC.
– We present a shallow embedding of EC types and judgments in LK. This both refines Nakano's treatment of effects and provides a pleasing interpretation of subtyping as logical consequence.

2 System EC

In this section we define system EC. EC is a call-by-value lambda calculus extended with exceptions. Its syntax is defined in Figure 1. The syntax of types is non-standard. In particular, arrows have form $\tau \to \Delta$, where Δ is an entire result context (see Section 2.1).

Exceptions are declared in signatures, Σ. A signature entry $E \sim \tau$ introduces a new exception named E carrying data of type τ. (Signature names are drawn from a distinguished set of identifiers.) Additionally, signatures may introduce subtyping constraints of form $E_1 <: E_2$. During evaluation, exception handlers for E_2 also handle E_1 exceptions. Type preservation requires that exception subtyping be consistent: E_1's data must be a subtype of E_2's data. This constraint is enforced by the well-formed signature judgment, $\Sigma \vdash \diamond$.

Exceptions are first class, and exception values are created using application syntax: $E\ e$. Expressions of the form $\mathsf{raise}\ e$ raise an exception, and dynamically propagating exceptions are also represented using raise. The final exception-specific construct is $e_1\ \mathsf{handle}\ E\ x \Rightarrow e_2$. This expression evaluates e_1. If e_1 raises exception E_1 where $E_1 <: E$, then e_2 is evaluated with E_1's data substituted for x. Otherwise the handle expression evaluates to the result of e_1.

The formalization of EC's dynamic semantics (Figure 2) is straightforward. Reduction is based on shallow evaluation contexts called *frames*. In addition to call-by-value lambda calculus reductions, EC has evaluation rules for exceptions. Rules E-Handle and E-Prop handle and propagate exceptions as discussed above. Choosing which rule to apply depends on dynamic knowledge of the subtyping hierarchy. For this reason, the evaluation relation is parameterized by a signature. The signature is also used by judgment $\Sigma \vdash E \curvearrowright F$, pronounced "E hops F." This holds when frame F *does not* handle exception E. Note that for the special case of exception subtyping, a simple graph traversal suffices to decide $\Sigma \vdash \mathsf{Exn}\ E_1 <: \mathsf{Exn}\ E_2$—there is no reason to invoke the typechecker at runtime.

2.1 Static Semantics

The full static semantics for EC are given by Figure 3. There we use $x \sharp \Delta$ to denote that a variable x is fresh with respect to Δ.

The typing relation is written $e : \Sigma; \Gamma \vdash \Delta$. The judgment's unusual shape foreshadows the embedding we define in Section 3.1. *Result context* Δ may have multiple components, for instance $\mathsf{ret} : \tau_1, \mathsf{exn}\ E_2 : \tau_2, \mathsf{exn}\ E_3 : \tau_3$. This can be read

Result channels

$\varepsilon ::=$	ret	Return channel
$\mid$	$\mathsf{exn}\ E$	Exception channel

Types

$\tau ::=$	A	Type variable
$\mid$	$\tau \to \Delta$	Arrow
$\mid$	$\top$	Top
$\mid$	$\mathsf{Exn}\ E$	Exception type

Result contexts

$\Delta ::= \cdot \mid \Delta, \varepsilon : \tau$

Variable contexts

$\Gamma ::= \cdot \mid \Gamma, x : \tau$

Signatures

$\Sigma ::=$	$\cdot$	Empty signature
$\mid$	$\Sigma, E \sim \tau$	Exception declaration
$\mid$	$\Sigma, E_1 <: E_2$	Subtype declaration

Expressions

$e ::=$	$x \mid e_1\ e_2 \mid \lambda x\colon \tau.\ e$	Lambda calulus
$\mid$	top	Top/unit value
$\mid$	$E\ e$	Exception expression
$\mid$	$\mathsf{raise}\ e$	Throw exception
$\mid$	$e_1\ \mathsf{handle}\ E\ x \Rightarrow e_2$	Exception handler

Values

$v ::= \lambda x\colon \tau.\ e \mid E\ v \mid \mathsf{top}$

Frames

$F ::=$	$[\,]\ e \mid v\ [\,]$	CVB application
$\mid$	$\mathsf{raise}\ [\,] \mid E\ [\,]$	Exception construction
$\mid$	$[\,]\ \mathsf{handle}\ E\ x \Rightarrow e$	Exception handler

Fig. 1. The syntax of EC.

in two ways. The programmer's reading is, "e is program which either returns a τ_1 or raises one of exception E_1 or E_2." In contrast, the logician reads, "e is a proof that demonstrates the validity of τ_1, or τ_2, or τ_3." As we will see in Section 3.2, both are right!

The Typing Relation We will explain EC's typing rules from the programmer's perspective. Σ declares exceptions as discussed above, and Γ maps variables to their types. Δ is a list of *result channels*, each corresponding to either the result of evaluating e or a potentially uncaught exception. Note that Δ is not allowed to contain multiple occurrences of the same channel. (This is enforced by the well-formedness judgments).

T-Var and T-Top simply translate standard typing rules into our setting. Because such expressions always return, each only has a ret channel.

Rule T-Exn types expressions of form $E\ e$, which introduce first-class exception values of type $\mathsf{Exn}\ E$. As e may raise exceptions during evaluation, its exception channels must also appear in $E\ e$'s result context.

Rule T-Raise takes an exception of type $\mathsf{Exn}\ E$ and throws it. There is no way for $\mathsf{raise}\ e$ to simply return a value; T-Raise reflects this, as its conclusion does not contain a ret channel. The typing rule's premise assumes that e may throw E—this is a not a problem because subtyping will allow us to add such an exception channel when one is not otherwise present.

As usual, T-Abs types $\lambda x\colon \tau.\ e$ by augmenting the environment with $x\colon \tau$ and recursively checking e. However the function is assigned type $\tau \to \Delta$. Thus the arrow type constructor internalizes the *entire* result context. The logician

$\boxed{\Sigma \vdash E \curvearrowright F}$

$$\text{E-F-App1}\ \frac{}{\Sigma \vdash E \curvearrowright [\,]\ e} \qquad \text{E-F-App2}\ \frac{}{\Sigma \vdash E \curvearrowright v\ [\,]} \qquad \text{E-F-Exn}\ \frac{}{\Sigma \vdash E_1 \curvearrowright E_2\ [\,]} \qquad \text{E-F-Raise}\ \frac{}{\Sigma \vdash E \curvearrowright \mathsf{raise}\ [\,]}$$

$$\text{E-F-Handle}\ \frac{\Sigma \nvdash \mathsf{Exn}\ E_1 <: \mathsf{Exn}\ E_2}{\Sigma \vdash E_1\ \curvearrowright\ [\,]\ \mathsf{handle}\ E_2\ \Rightarrow e}$$

$\boxed{\Sigma \vdash e \to e'}$

$$\text{E-Beta}\ \frac{}{\Sigma \vdash (\lambda x\colon \tau.\ e)\ v \to e\{v/x\}} \qquad \text{E-Handle}\ \frac{\Sigma \vdash \mathsf{Exn}\ E_1 <: \mathsf{Exn}\ E_2}{\Sigma \vdash (\mathsf{raise}\ E_1\ v)\ \mathsf{handle}\ E_2\ x \Rightarrow e \to e\{v/x\}}$$

$$\text{E-Propagate}\ \frac{\Sigma \vdash E \curvearrowright F}{\Sigma \vdash F[\mathsf{raise}\ E\ v] \to \mathsf{raise}\ E\ v} \qquad \text{E-Frame}\ \frac{\Sigma \vdash e \to e'}{\Sigma \vdash F[e] \to F[e']}$$

Fig. 2. Evaluation relation for EC.

may find this unexpected; perhaps he was anticipating something more like LK-ImpR from Figure 5. The programmer may be less surprised. The exception channels on the right of an arrow are *latent effects*, which are delayed until the corresponding function is applied [6].

Application behaves mostly as expected. T-App's premises mention three result contexts which must contain identical exception channels. As with T-Raise, this may be accomplished using subtyping.

Rule T-Handle checks e_1 $\mathsf{handle}\ E\ x \Rightarrow e_2$ as follows. Expression e_1's result context is partitioned into Δ_1, Δ_2 where Δ_1 is made of exceptions beneath E in the subtype relation. That is, Δ_1 is the set of caught exceptions. Checking e_2 (in a suitably augmented environment) must then yield Δ_2. As both e_1 and e_2 produce Δ_2s, the two expressions must agree on ret channels (if any). Exceptions allowed in Δ_1 are also allowed in Δ_2, which is useful if e_2 raises exceptions.

The final typing rule, T-Sub casts expressions to supertypes. As we have seen, the availability of subtyping is used to give rules like T-App a simple form.

Subtyping and Well-formedness EC contains three subtyping relations. These relate pairs of types, pairs of result channels, and pairs of result contexts.

The relation on types contains several standard rules familiar from system $F_{<:}$: S-Refl, S-Trans, and S-Top. The arrow rule is contravariant on arguments and covariant on results. But this is not completely standard as the results are contexts, not basic types.

Exception subtyping is nominal and is governed by signature declarations. Signature well-formedness ensures that all declared subtypes respect covariance. EC does not require that there be a top exception—a supertype of all other exceptions like Java's throwable. Of course, such an exception may be defined.

Result context subtyping is more interesting. Intuitively, $\Sigma \vdash \Delta_1 <: \Delta_2$ when program contexts that handle all channels in Δ_2 also handle all channels in Δ_1. Each of Δ_1's exception channels must be a subtype of an exception channel in Δ_2, and Δ_1's return channel must be a subtype of Δ_2's. For example, consider the following well-formed signature.

$$\Sigma = \mathsf{Disk} \sim \mathsf{string}, \mathsf{Sound} \sim \top, \mathsf{IO} \sim \top,$$
$$\mathsf{Disk} <: \mathsf{IO}, \mathsf{Sound} <: \mathsf{IO}.$$

Then the following judgments hold.

$$\Sigma \vdash \mathsf{exn\ Disk} : \mathsf{string}, \mathsf{exn\ Sound} : \top <: \mathsf{exn\ IO} : \top$$

$$\Sigma \vdash \mathsf{exn\ Sound} : \top <: \mathsf{exn\ Sound} : \top, \mathsf{exn\ Disk} : \mathsf{string}$$

But these cannot be derived.

$$\Sigma \nvdash \mathsf{exn\ Disk} : \mathsf{string}, \mathsf{exn\ Sound} : \top <: \mathsf{exn\ Disk} : \mathsf{string}$$

$$\Sigma \nvdash \mathsf{ret} : \mathsf{string} <: \mathsf{exn\ IO} : \top$$

The programmer recognizes that result context subtyping lets him "over-annotate" an expression's effects—a familiar operation in type and effect systems [16]. The logician views this as a form of hypothetical reasoning: $\Delta_1 <: \Delta_2$ when the disjunction of Δ_1's formulas implies the disjunction of Δ_2's. The logician's view is made rigorous by Theorem 1.

Result channel subtyping ($\Sigma \vdash \varepsilon_1 : \tau_1 <: \varepsilon_2 : \tau_2$) is simple: return channels admit subtyping, but exception channels do not. It appears the latter restriction could be relaxed, but doing so does not provide an obvious advantage. Furthermore, the restriction helps maintain a useful invariant: each exception channel is annotated by the exception's declared type (and not a subtype).

EC also includes several well-formedness judgments. These are needed to maintain three invariants. First, signatures, environments, and result contexts do not bind the same element twice. Second, exception references (e.g., in types) only name previously defined exceptions. Third, as mentioned above, subtype declarations are consistent with structural subtyping.

2.2 Metatheory

This section provides a brief overview of the main properties of EC.

The type system is set up such that the well-formedness of signatures and environments is checked at the leaves of typing derivations. Consequently EC enjoys a variety of regularity lemmas like the following.

$\boxed{e : \Sigma; \Gamma \vdash \Delta}$

$$\text{T-Sub}\ \frac{e : \Sigma; \Gamma \vdash \Delta_0 \qquad \Sigma \vdash \Delta_0 <: \Delta}{e : \Sigma; \Gamma \vdash \Delta} \qquad \text{T-Var}\ \frac{(x : \tau) \in \Gamma \qquad \Sigma \vdash \Gamma}{x : \Sigma; \Gamma \vdash \mathsf{ret} : \tau} \qquad \text{T-Top}\ \frac{\Sigma \vdash \Gamma}{\mathsf{top} : \Sigma; \Gamma \vdash \mathsf{ret} : \top}$$

$$\text{T-Abs}\ \frac{e : \Sigma; \Gamma, x : \tau \vdash \Delta}{\lambda x : \tau.\ e : \Sigma; \Gamma \vdash \mathsf{ret} : \tau \to \Delta}$$

$$\text{T-App}\ \frac{e_1 : \Sigma; \Gamma \vdash \mathsf{ret} : \tau_2 \to (\mathsf{ret} : \tau, \Delta), \Delta \qquad e_2 : \Sigma; \Gamma \vdash \mathsf{ret} : \tau_2, \Delta}{e_1\ e_2 : \Sigma; \Gamma \vdash \mathsf{ret} : \tau, \Delta}$$

$$\text{T-Exn}\ \frac{e : \Sigma; \Gamma \vdash \mathsf{ret} : \tau, \Delta \qquad E \sim \tau \in \Sigma}{E\ e : \Sigma; \Gamma \vdash \mathsf{ret} : \mathsf{Exn}\ E, \Delta} \qquad \text{T-Raise}\ \frac{e : \Sigma; \Gamma \vdash \mathsf{ret} : \mathsf{Exn}\ \tau, \Delta, \mathsf{exn}\ E : \tau}{\mathsf{raise}\ e : \Sigma; \Gamma \vdash \Delta, \mathsf{exn}\ E : \tau}$$

$$\text{T-Handle}\ \frac{\Sigma \vdash \Delta_1 <: \cdot, \mathsf{exn}\ E : \tau \qquad e_1 : \Sigma; \Gamma \vdash \Delta_1, \Delta_2 \qquad e_2 : \Sigma; \Gamma, x : \tau \vdash \Delta_2}{e_1\ \mathsf{handle}\ E\ x \Rightarrow e_2 : \Sigma; \Gamma \vdash \Delta_2}$$

$\boxed{\Sigma \vdash \tau_1 <: \tau_2}$

$$\text{S-Refl}\ \frac{\Sigma \vdash \tau}{\Sigma \vdash \tau <: \tau} \qquad \text{S-Trans}\ \frac{\Sigma \vdash \tau_1 <: \tau_2 \qquad \Sigma \vdash \tau_2 <: \tau_3}{\Sigma \vdash \tau_1 <: \tau_3}$$

$$\text{S-Top}\ \frac{\Sigma \vdash \tau}{\Sigma \vdash \tau <: \top} \qquad \text{S-ExnT}\ \frac{\Sigma \vdash \diamond \qquad E_1 <: E_2 \in \Sigma}{\Sigma \vdash \mathsf{Exn}\ E_1 <: \mathsf{Exn}\ E_2} \qquad \text{S-Arr}\ \frac{\Sigma \vdash \tau_2 <: \tau_1 \qquad \Sigma \vdash \Delta_1 <: \Delta_2}{\Sigma \vdash \tau_1 \to \Delta_1 <: \tau_2 \to \Delta_2}$$

$\boxed{\Sigma \vdash \Delta_1 <: \Delta_2}$

$$\text{S-Nil}\ \frac{\Sigma \vdash \Delta}{\Sigma \vdash \cdot <: \Delta} \qquad \text{S-Cons}\ \frac{\Sigma \vdash \Delta_1 <: \Delta_2 \qquad \varepsilon_2 : \tau_2 \in \Delta_2 \qquad \Sigma \vdash \varepsilon_1 : \tau_1 <: \varepsilon_2 : \tau_2}{\Sigma \vdash \Delta_1, \varepsilon_1 : \tau_1 <: \Delta_2}$$

$\boxed{\Sigma \vdash \varepsilon_1 : \tau_1 <: \varepsilon_2 : \tau_2}$

$$\text{S-Ret}\ \frac{\Sigma \vdash \tau_1 <: \tau_2}{\Sigma \vdash \mathsf{ret} : \tau_1 <: \mathsf{ret} : \tau_2}$$

$$\text{S-ExnC}\ \frac{\Sigma \vdash \mathsf{Exn}\ E_1 <: \mathsf{Exn}\ E_2 \qquad \Sigma \vdash \mathsf{exn}\ E_1 : \tau_1 \qquad \Sigma \vdash \mathsf{exn}\ E_2 : \tau_2}{\Sigma \vdash \mathsf{exn}\ E_1 : \tau_1 <: \mathsf{exn}\ E_2 : \tau_2}$$

Fig. 3. Static Semantics for EC. (1/2)

$\boxed{\Sigma \vdash \diamond}$

$$\textsc{Wf-S-Nil}\ \frac{}{\cdot \vdash \diamond} \qquad \textsc{Wf-S-Decl}\ \frac{\Sigma \vdash \tau}{\Sigma, E \sim \tau \vdash \diamond}\ \text{where } E \sharp \Sigma$$

$$\textsc{Wf-S-Sub}\ \frac{E_1 \sim \tau_1 \in \Sigma \qquad E_2 \sim \tau_2 \in \Sigma \qquad \Sigma \vdash \tau_1 <: \tau_2}{\Sigma, E_1 <: E_2 \vdash \diamond}$$

$\boxed{\Sigma \vdash \tau}$

$$\textsc{Wf-T-Var}\ \frac{\Sigma \vdash \diamond}{\Sigma \vdash A} \qquad \textsc{Wf-T-Top}\ \frac{\Sigma \vdash \diamond}{\Sigma \vdash \top} \qquad \textsc{Wf-T-Arr}\ \frac{\Sigma \vdash \tau \qquad \Sigma \vdash \Delta}{\Sigma \vdash \tau \to \Delta} \qquad \textsc{Wf-T-Exn}\ \frac{\Sigma \vdash \diamond \qquad E \sim \tau \in \Sigma}{\Sigma \vdash \mathsf{Exn}\ E}$$

$\boxed{\Sigma \vdash \Delta}$

$$\textsc{Wf-C-Nil}\ \frac{\Sigma \vdash \diamond}{\Sigma \vdash \cdot} \qquad \textsc{Wf-C-Cons}\ \frac{\Sigma \vdash \Delta \qquad \Sigma \vdash \varepsilon : \tau}{\Sigma \vdash \Delta, \varepsilon : \tau}\ \text{where } \varepsilon \sharp \Delta$$

$\boxed{\Sigma \vdash \varepsilon : \tau}$

$$\textsc{Wf-B-Ret}\ \frac{\Sigma \vdash \tau}{\Sigma \vdash \mathsf{ret} : \tau} \qquad \textsc{Wf-B-Exn}\ \frac{\Sigma \vdash \diamond \qquad E \sim \tau \in \Sigma}{\Sigma \vdash \mathsf{exn}\ E : \tau}$$

$\boxed{\Sigma \vdash \Gamma}$

$$\textsc{Wf-E-Nil}\ \frac{\Sigma \vdash \diamond}{\Sigma \vdash \cdot} \qquad \textsc{Wf-E-Cons}\ \frac{\Sigma \vdash \Gamma \qquad \Sigma \vdash \tau}{\Sigma \vdash \Gamma, x : \tau}\ \text{where } x \sharp \Gamma$$

Fig. 4. Static Semantics for EC. (2/2)

Lemma 1. *Suppose* $\mathcal{D} :: \Sigma \vdash \tau_1 <: \tau_2$. *Then a subderivation of* $\mathcal{D}$ *shows* $\Sigma \vdash \diamond$.

Proof (Proof Sketch). Proof is by simultaneous induction with the following proposition: $\mathcal{D} :: \Sigma \vdash \tau$ implies that a subderivation of $\mathcal{D}$ shows $\Sigma \vdash \diamond$.

We state Lemma 1 in terms of subderivations to facilitate proving Lemma 9 by structural induction.

Because EC includes subtyping, inverting typing judgments provides little traction for reasoning. We must prove a variety of canonical forms and inversion properties. One such lemma follows.

Lemma 2 (Cannonical Form—Abs). *Assume*

$$v : \Sigma; \Gamma \vdash \mathsf{ret} : \tau_2 \to (\mathsf{ret} : \tau_1, \Delta_1), \Delta$$

then $v = \lambda x\colon \tau_3.\ e$ *where* $\Sigma \vdash \tau_2 <: \tau_3$.

Proof. By induction on the typing derivation, using an inner induction for the T-Sub case.

The proof of progress is straightforward, but preservation requires several auxiliary lemmas. Lemma 5 is particularly interesting. It states that values can always be typed in a result context containing only a ret channel. Intuitively this corresponds to the fact that values do not reduce and therefore cannot raise exceptions. Lemmas **??**–6 establish useful invariants of exception and frame typing.

Lemma 3 (Progress). *Assume* $e\colon \Sigma; \cdot \vdash \Delta$. *Then either* e *is value,* e *has form* $\mathsf{raise}\ E\ v$ *(i.e., is an uncaught exception), or* $\Sigma \vdash e \to e'$.

Proof. Proof by induction on the typing derivation. Case T-App uses Lemma 2.

Lemma 4 (Substitution). *Suppose* $e : \Sigma; \Gamma, x\colon \tau, \Gamma' \vdash \Delta$ *and* $v : \Sigma; \Gamma, \Gamma' \vdash \mathsf{ret}\colon \tau$, *then* $e\{v/x\}\colon \Sigma; \Gamma, \Gamma' \vdash \Delta$.

Lemma 5 (Value Strengthening). *Suppose* $v\colon \Sigma; \Gamma \vdash \mathsf{ret}\colon \tau, \Delta$, *then* $v\colon \Sigma; \Gamma \vdash \mathsf{ret}\colon \tau$

Proof. Lemmas 4 and 5 follow from straightforward induction.

Lemma 6 (Unhandled Exception Frame Typing). *Suppose* $\Sigma \vdash E \curvearrowright F$ *and* $F[\mathsf{raise}\ E\ v]\colon \Sigma; \Gamma \vdash \Delta$. *Then* $\mathsf{exn}\ E\colon \tau \in \Delta$ *where* $E \sim \tau \in \Sigma$.

Proof. Direct using elided inversion principles.

Lemma 7 (Preservation). *Assume* $\Sigma \vdash e \to e'$. *For all* Δ *such that* $e\colon \Sigma; \cdot \vdash \Delta$, *it follows that* $e'\colon \Sigma; \cdot \vdash \Delta$.

Proof. By induction on the reduction relation.

3 The logical content of EC programs

Thus far we have discussed EC from the perspective of a programmer, as a language with a sound, type-based exception analysis. This section shows that EC has a logical reading, namely typing derivations in EC LK

To begin, we review LK. LK is a logic which operates on propositions drawn from the following grammar.

LK Propositions
$$P, Q ::= A_{\mathrm{LK}} \mid \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P$$

Atomic propositions, A_{LK} are subscripted to avoid confusion with EC types. LK judgments have the form $\Phi \vdash \Theta$, where Φ—the *antecedent*—and Θ—the *succedent*—are lists of propositions. Intuitively, $\Phi \vdash \Theta$ holds when the conjunction of the Φs implies the disjunction of the Θs. The rules of deduction for propositional LK are given by Figure 5.

$\boxed{\Phi \vdash \Theta}$

$$\textsc{LK-Id}\ \frac{}{P \vdash P} \qquad \textsc{LK-ThinL}\ \frac{\Phi \vdash \Theta}{P, \Phi \vdash \Theta} \qquad \textsc{LK-ThinR}\ \frac{\Phi \vdash \Theta}{\Phi \vdash \Theta, P} \qquad \textsc{LK-ContractL}\ \frac{P, P, \Phi \vdash \Theta}{P, \Phi \vdash \Theta} \qquad \textsc{LK-ContractR}\ \frac{\Phi \vdash \Theta, P, P}{\Phi \vdash \Theta, P}$$

$$\textsc{LK-InterL}\ \frac{\Phi_1, Q, P, \Phi_2 \vdash \Theta}{\Phi_1, P, Q, \Phi_2 \vdash \Theta} \qquad \textsc{LK-InterR}\ \frac{\Phi \vdash \Theta_1, Q, P, \Theta_2}{\Phi \vdash \Theta_1, P, Q, \Theta_2} \qquad \textsc{LK-Cut}\ \frac{\Phi_1 \vdash \Theta_1, P \qquad P, \Phi_2 \vdash \Theta_2}{\Phi_1, \Phi_2 \vdash \Theta_1, \Theta_2}$$

$$\textsc{LK-AndL1}\ \frac{P, \Phi \vdash \Theta}{P \wedge Q, \Phi \vdash \Theta} \qquad \textsc{LK-AndR}\ \frac{\Phi \vdash \Theta, P \qquad \Phi \vdash \Theta, Q}{\Phi \vdash \Theta, P \wedge Q} \qquad \textsc{LK-AndL2}\ \frac{Q, \Phi \vdash \Theta}{P \wedge Q, \Phi \vdash \Theta} \qquad \textsc{LK-OrR1}\ \frac{\Phi \vdash \Theta, P}{\Phi \vdash \Theta, P \vee Q}$$

$$\textsc{LK-OrL}\ \frac{P, \Phi \vdash \Theta \qquad Q, \Phi \vdash \Theta}{P \vee Q, \Phi \vdash \Theta} \qquad \textsc{LK-OrR2}\ \frac{\Phi \vdash \Theta, Q}{\Phi \vdash \Theta, P \vee Q} \qquad \textsc{LK-ImpL}\ \frac{\Phi_1 \vdash \Theta_1, P \qquad Q, \Phi_2 \vdash \Theta_2}{P \supset Q, \Phi_1, \Phi_2 \vdash \Theta_1, \Theta_2}$$

$$\textsc{LK-ImpR}\ \frac{P, \Phi \vdash \Theta, Q}{\Phi \vdash \Theta, P \supset Q} \qquad \textsc{LK-NotL}\ \frac{\Phi \vdash \Theta, P}{\neg P, \Phi \vdash \Theta} \qquad \textsc{LK-NotR}\ \frac{P, \Phi \vdash \Theta}{\Phi \vdash \Theta, \neg P}$$

$$\textsc{LK-FalseL}\ \frac{}{\bot \vdash} \qquad \textsc{LK-TrueR}\ \frac{}{\vdash \top}$$

Fig. 5. Validity judgement for LK.

3.1 An embedding of EC types in LK

We show that every EC typing judgment can be translated into a valid LK judgment. The translation is defined by four functions. $[\![\tau]\!]^{\Sigma}_{\mathsf{typ}}$ maps types to propositions, $[\![\Delta]\!]^{\Sigma}_{\mathsf{ctx}}$ maps EC result contexts into LK succedents, and $[\![\Gamma]\!]^{\Sigma}_{\mathsf{env}}$ maps EC environments into LK antecedents. Lastly $\Cup\Theta$, pronounced "lift Θ," internalizes Θ's commas as disjunctions.

The type-to-proposition translation is defined as follows.

$$\begin{aligned}
[\![A]\!]^{\Sigma}_{\mathsf{typ}} &= A_{\mathrm{LK}} \\
[\![\tau \to \Delta]\!]^{\Sigma}_{\mathsf{typ}} &= [\![\tau]\!]^{\Sigma}_{\mathsf{typ}} \supset \Cup [\![\Delta]\!]^{\Sigma}_{\mathsf{ctx}} \\
[\![\top]\!]^{\Sigma}_{\mathsf{typ}} &= \top \\
[\![\mathsf{Exn}\ E]\!]^{\Sigma}_{\mathsf{typ}} &= [\![\tau]\!]^{\Sigma}_{\mathsf{typ}} \qquad \text{where } E \sim \tau \in \Sigma
\end{aligned}$$

We assume that there is an atomic LK proposition for each EC type variable. The interesting cases are for arrows and exceptions. The translation of $\tau \to \Delta$ can't simply return $[\![\tau]\!]^{\Sigma}_{\mathsf{typ}} \supset [\![\Delta]\!]^{\Sigma}_{\mathsf{ctx}}$—it's not well-formed. Instead we take the disjunction of all elements in $[\![\Delta]\!]^{\Sigma}_{\mathsf{ctx}}$. This follows the intuition that succedents

correspond to lists of disjunctions. The translation of an exception is the translation of the type it carries.

The environment and result context translations are simple.

$$\begin{aligned}\llbracket \cdot \rrbracket^{\Sigma}_{\mathsf{env}} &= \cdot \\ \llbracket \Gamma, x \colon \tau \rrbracket^{\Sigma}_{\mathsf{env}} &= \llbracket \Gamma \rrbracket^{\Sigma}_{\mathsf{env}}, \llbracket \tau \rrbracket^{\Sigma}_{\mathsf{typ}}\end{aligned}$$

$$\begin{aligned}\llbracket \cdot \rrbracket^{\Sigma}_{\mathsf{ctx}} &= \cdot \\ \llbracket \Delta, \varepsilon \colon \tau \rrbracket^{\Sigma}_{\mathsf{ctx}} &= \llbracket \Delta \rrbracket^{\Sigma}_{\mathsf{ctx}}, \llbracket \tau \rrbracket^{\Sigma}_{\mathsf{typ}}\end{aligned}$$

When lifting LK antecedents we do a little extra work to avoid unnecessary occurrences of $\bot$; this leads to an aesthetically cleaner translation.

$$\begin{aligned}\Cup \cdot &= \bot \\ \Cup \cdot, P &= P \\ \Cup \Theta, P &= \Cup \Theta \vee P \quad \text{where } \Theta \neq \cdot\end{aligned}$$

The $\Cup$ function internalizes result context commas as $\vee$s. While this is a syntactically well defined operation, it's not a priori clear that it preserves logical validity. The following lemma shows that it does. Linear logicians should note that by internalizing the succedent's comma we are, here, viewing $\vee$ as the multiplicative disjunction, $\parr$. This is noteworthy as embedded result contexts are operationally more similar to sequences of additive disjunctions, $\oplus$.

Lemma 8. *$\Phi \vdash \Theta$ iff $\Phi \vdash \Cup \Theta$.*

Proof. Proof by induction on the length of Θ.

3.2 Main Results

This section presents our main results. Lemma 9 shows that EC's subtyping relations corresponds to logical consequence in LK. And Theorem 1 shows this for EC typing judgments.

Lemma 9 (Subtyping Lemma). *Suppose $\mathcal{D}$ a subtyping derivation in EC.*

- *If $\mathcal{D} :: \Sigma \vdash \Delta_1 <: \Delta_2$ then $\Cup \llbracket \Delta_1 \rrbracket^{\Sigma}_{\mathsf{ctx}} \vdash \llbracket \Delta_2 \rrbracket^{\Sigma}_{\mathsf{ctx}}$.*
- *If $\mathcal{D} :: \Sigma \vdash \tau_1 <: \tau_2$ then $\llbracket \tau_1 \rrbracket^{\Sigma}_{\mathsf{typ}} \vdash \llbracket \tau_2 \rrbracket^{\Sigma}_{\mathsf{typ}}$.*
- *If $\mathcal{D} :: \Sigma \vdash \diamond$ then $\llbracket \tau_1 \rrbracket^{\Sigma}_{\mathsf{typ}} \vdash \llbracket \tau_2 \rrbracket^{\Sigma}_{\mathsf{typ}}$ where $E_1 <: E_2$, $E_1 \sim \tau_1$, $E_2 \sim \tau_2 \in \Sigma$.*

Proof. Proof is by structural induction on $\mathcal{D}$. We will show only the most interesting cases. Begin by examining the cases for $\Sigma \vdash \tau_1 <: \tau_2$.

- Case S-Trans: Immediate from the induction hypothesis and LK-Cut.

– Case S-Arr: The subtyping derivation ends in

$$\frac{\Sigma \vdash \tau_2 <: \tau_1 \qquad \Sigma \vdash \Delta_1 <: \Delta_2}{\Sigma \vdash \tau_1 \to \Delta_1 <: \tau_2 \to \Delta_2}$$

We want to show

$$\left([\![\tau_1]\!]^\Sigma_{\mathsf{typ}} \supset \Cirv[\![\Delta_1]\!]^\Sigma_{\mathsf{ctx}}\right) \vdash \left([\![\tau_2]\!]^\Sigma_{\mathsf{typ}} \supset \Cirv[\![\Delta_2]\!]^\Sigma_{\mathsf{ctx}}\right).$$

The induction hypothesis gives $[\![\tau_2]\!]^\Sigma_{\mathsf{typ}} \vdash [\![\tau_1]\!]^\Sigma_{\mathsf{typ}}$ and $\Cirv[\![\Delta_1]\!]^\Sigma_{\mathsf{ctx}} \vdash [\![\Delta_2]\!]^\Sigma_{\mathsf{ctx}}$. Lemma 8 transforms the latter to $\Cirv[\![\Delta_1]\!]^\Sigma_{\mathsf{ctx}} \vdash \Cirv[\![\Delta_2]\!]^\Sigma_{\mathsf{ctx}}$. Conclude with an easy LK derivation.

Cases for $\Sigma \vdash \diamond$.

– Case Wf-S-Sub: Suppose Σ ends with $E_3 <: E_4$. Conclude with the induction hypothesis, using Lemma 1 if $E_1 \neq E_3$ or $E_2 \neq E_4$.

Cases for $\Sigma \vdash \Delta_1 <: \Delta_2$

– Case S-Cons. The subcontext derivation ends in

$$\frac{\Sigma \vdash \Delta_1 <: \Delta_2 \qquad \varepsilon_2 : \tau_2 \in \Delta_2 \qquad \Sigma \vdash \varepsilon_1 : \tau_1 <: \varepsilon_2 : \tau_2}{\Sigma \vdash \Delta_1, \varepsilon_1 : \tau_1 <: \Delta_2}$$

First establish $[\![\tau_1]\!]^\Sigma_{\mathsf{typ}} \vdash [\![\tau_2]\!]^\Sigma_{\mathsf{typ}}$ and $\Cirv[\![\Delta_1]\!]^\Sigma_{\mathsf{ctx}} \vdash [\![\Delta_2]\!]^\Sigma_{\mathsf{ctx}}$ via inversion and appeal to the induction hypotheses. Next build this LK derivation:

$$\frac{\Cirv[\![\Delta_1]\!]^\Sigma_{\mathsf{ctx}} \vdash [\![\Delta_2]\!]^\Sigma_{\mathsf{ctx}} \qquad \dfrac{\dfrac{[\![\tau_1]\!]^\Sigma_{\mathsf{typ}} \vdash [\![\tau_2]\!]^\Sigma_{\mathsf{typ}}}{\text{several uses of LK-ThinR}}}{[\![\tau_1]\!]^\Sigma_{\mathsf{typ}} \vdash [\![\Delta_2]\!]^\Sigma_{\mathsf{typ}}}}{\Cirv[\![\Delta_1]\!]^\Sigma_{\mathsf{ctx}} \vee [\![\tau_1]\!]^\Sigma_{\mathsf{typ}} \vdash [\![\Delta_2]\!]^\Sigma_{\mathsf{ctx}}}$$

When Δ_1 is non-empty, we're done. Otherwise, we conclude with a simple LK derivation.

Theorem 1 (Main Theorem). *Suppose* $e \colon \Sigma; \Gamma \vdash \Delta$. *Then* $[\![\Gamma]\!]^\Sigma_{\mathsf{env}} \vdash [\![\Delta]\!]^\Sigma_{\mathsf{ctx}}$.

Proof. Proof is by induction on the typing derivation. Cases T-Sub and T-Handle use Lemmas 8 and 9.

4 Discussion

On call-by-value evaluation Because EC evaluation follows a call-by-value evaluation strategy we can use arrow typing rules that describe latent effects as suspended. This is advantageous from a programming perspective—it enables

precise reasoning about the sequencing of effects. However, this evaluation strategy cannot correspond to strong-normalization of LK; abstractions may hide applications which translate to LK cuts. Reduction in systems studied by Nakano [12] and others does correspond to strong-normalization, but these languages lack EC's simple and precise reasoning for call-by-value exceptions.

EC's call-by-value reduction makes type-and-effect-inspired analysis far more appealing than a monadic effect system. Consider the following program.

$$(\lambda x\colon \top.\ \mathsf{top})\ (\mathsf{raise\ Fail\ "disaster!"})$$

EC types this with $\mathsf{ret}\colon \top, \mathsf{exn\ Fail}\colon \mathsf{string}$. This is safe and (reasonably) precise for a strict language; evaluation will raise the reported exception.

Call-by-name throws away the raise expression and no exception is thrown. Haskell's monadic effect discipline accounts for this in a precise way. The raise expression can be assigned type $\mathsf{M\ b}$, where M is the exception monad, and the function gets type $\mathsf{forall\ a}, \mathsf{a} \to \top$. (We have to squint a bit because EC lacks polymorphism and Haskell lacks subtyping.) The entire expression has type $\top$—a sound result given call-by-name evaluation.

However, type $\top$ is unsound for call-by-value evaluation. What goes wrong is that unused function parameters are relevant to evaluation, but not to the monadic effects system. It appears possible to cook the monadic analysis so that it's sound for call-by-value. One way to do this is by imposing side conditions on the abstraction or application typing rules. However such an invasive change is unsatisfying. It negates one of the key advantages of monad-based effect analysis in Haskell: independence of effect monads from the rest of the type system.

Open questions There are several more questions we can ask about EC itself and its relation to LK: What is the right way to add polymorphism to EC? Is EC strongly normalizing? How can EC extensions deal with world effects like memory access?

What is the logical content of EC proofs? EC is weaker than classical logical, perhaps even strictly weaker. It appears obvious that EC is as powerful as intuitionistic logic. Can the shallow embedding in this paper be extended to an isomorphism? Should such an isomorphism be with LK, a Full Intuitionistic Logic Kleene [8], or something else?

5 Related Work

Logics with control Griffin [4] first established that proofs-as-program correspondence between lambda calculus and intuitionistic logic could be extended to control effects and classical logic. Subsequently, several authors have investigated the relation between control effects, classical logic, and sequent calculi.

Nakano [13] observed a correlation between exception analysis and LK, a classical logic. His system $L_{c/t}$ uses lexically scoped variables, or "tags," as identifiers for exception tracking. However the tags-as-variables style requires many administrative tag abstractions and instantiations to avoid problems stemming

from variable confusion. Additionally, Nakano's static semantics fails to directly account for latent effects. His notion of "compatible contexts" hints at subtyping, but does explore it substantially.

Ong and Stewart [14] extend Parigot's $\lambda\mu$ calculus in their μPCF system and propose an encoding of ML-like exceptions similar to EC's. Key differences are that μPCF identifies exception names with covariables and does not address exception subtyping.

De Groote [2] examines an exception analysis based on classical natural deduction. Exceptions that carry type τ have type $\tau \to \bot$, and classical reasoning principals may be encoded in his system. The system uses lexically scoped variables as exception names and requires a nonstandard reduction scheme to prevent these names from escaping their scope. The analysis only types programs that do not raise uncaught exceptions and does not account for latent effects.

Several authors have encoded exceptions and exception analysis with control operators. As illustrated by Lillibridge [10], the exact relation between these features is highly sensitive to the surrounding language. We choose to define exceptions directly as definition by elaboration fails to provide programmers with an intuitive language specification.

The type-and-effect discipline for exceptions Type-and-effects systems track side effects in (usually) call-by-value languages [15, 6] Typically this uses a judgment of the form $\Gamma \vdash e : \tau \,!\, \ell$. *Label* ℓ describes the set of effects which may be triggered when evaluating e. Additionally, arrow types are annotated with latent effects.

Leroy and Pessaux [9] use an effects system to detect uncaught exceptions in ML programs. (Research by Guzmán and Suárez [5] foreshadows this work.) This analysis differs from ours as it considers ML-style exceptions with no subtyping, and attempts to track values carried by exceptions. Blume et al. [1] use a refinement of Leroy's techniques to analyze exceptions in a language with extensible cases, showing similarity between the exception and case mechanisms. Kennedy [7] defines a type system for a CPS-style intermediate language, and observes that type assignment for the exception continuation corresponds to an effect analysis.

6 Conclusion

The paper introduces EC, a language with a sound, typed-based analysis for Java-style exceptions. We show that EC's types and static judgments can be embedded in Gentzen's LK. This clarifies the relation between effects analysis, an important enabling technology for compilation and verification, and classical sequent calculi, an interesting and well-understand family of logics.

Acknowledgments I thank Steve Zdancewic for his helpful feedback on drafts of this paper, and Hugo Herbelin and Guillaume Munch-Maccagnoni for their discussions about classical logic.

Bibliography

[1] M. Blume, U. Acar, and W. Chae. Exception handlers as extensible cases. In *Proc. APLAS '08*, 2008.

[2] P. de Groote. A simple calculus of exception handling. In *Proc. TLCA '95*. Springer-Verlag, 1995.

[3] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1935. Collection 1969.

[4] T. Griffin. A formulae-as-type notion of control. In *POPL '90*. ACM, 1990.

[5] J. C. Guzmán and A. Suárez. A type system for exceptions. In *Proc. of the 1994 workshop on ML and its applications*, 1994.

[6] P. Jouvelot and D. Gifford. Reasoning about continuations with control effects. *SIGPLAN Not.*, 24(7):218–226, 1989.

[7] A. Kennedy. Compiling with continuations, continued. In *Proc. ICFP '07*. ACM, 2007.

[8] Stephen Kleene. *Introduction to Metamathematics*. D. Van Nostrand Company, Inc., New York, 1952.

[9] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.

[10] M. Lillibridge. Exceptions are strictly more powerful than call/cc. Technical Report CMU-CS-95-178, Carnegie Mellon University, July 1995.

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

[12] H. Nakano. A constructive formalization of the catch and throw mechanism. In *LICS '92*, pages 82–89, 1992.

[13] H. Nakano. The non-deterministic catch and throw mechanism and its subject reduction property. In *Logic, Language and Computation, Festschrift in Honor of Satoru Takasu*, pages 61–72, 1994.

[14] L. Ong and C. Stewart. A Curry-Howard foundation for functional computation with control. In *POPL '97*, pages 215–227, 1997.

[15] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *LICS'92*. IEEE Press, June 1992.

[16] G. Washburn. Cause and effect: type systems for effects and dependencies. Technical Report MS-CIS-05-05, University of Pennsylvania, July 2005.