One Plus One: Effect System as a Synergy of Implicit Parameters and Type-Based Escape Analysis

ANDREY STOYAN, HSE University, Russian Federation

Effects offer a systematic approach to managing code complexity. Certain responsibilities can be delegated to an execution context, allowing a computation to interact with this context by performing effects and thereby focusing on other aspects of logic. Effect systems elevate information about a program's side effects to the type level, clarifying abstraction boundaries and enabling static verification of context validity. Despite these advantages, effect systems have not seen widespread adoption, largely due to their alien design and the general unfamiliarity of practitioners with the concept.

In this paper, we propose a perspective on effect system design as a combination of two properly-designed language features: implicit parameters and type-based escape analysis. We argue that both features can be naturally realized in mainstream programming languages, are already familiar to practitioners, and provide independent utility. We motivate our approach by considering effect systems as mechanism for tracking free variables. We specify the requirements for a proper design of implicit parameters and illustrate these requirements through a formal example. We introduce a novel type-based escape analysis technique based on polymorphic λ-calculus with subtyping that relies on straightforward programmer-supplied dataflow descriptions. Furthermore, we combine implicit parameters and type-based escape analysis together to provide an example of a minimalistic effect system. Finally, we discuss technical approaches to make an effect system more practical for common programming scenarios.

CCS Concepts: • **Software and its engineering → Polymorphism**; *Control structures*; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: static analysis, type systems, effects, capabilities

1 Introduction

The discipline of programming primarily centers on the management of complexity, enabling developers to concentrate on a specific fragment of behavior within any given code segment. Abstraction and the separation of concerns are essential strategies in this process: distinct responsibilities are delegated to different program entities, such as functions or modules. Frequently, such an entity may be abstractly referred to as the *execution context* of a computation, with *effects* emerging naturally as interactions between the computation and its context [11].

An execution context can be characterized by at least one of the following properties:

(1) Interactions with an execution context are observable. For example, the memory management subsystem can be considered as the execution context — each memory mutation can influence the results of subsequent reads. In that case the context is responsible for bookkeeping of memory cells.
(2) The activity of an execution context is restricted to a particular scope, such that only code within this scope can interact with the context. For example, an exception handler operates as an execution context confined to the body of a `try-catch` block; when an exception is thrown, control is transferred to the handler, which manages the exceptional situation. Another case is inversion of control, where an execution context restricts the provision of certain functionality to code within a defined scope, while code outside that scope may interact with a different context and access different functionality.

An effect system reflects effects of a computation at the type level. Following the properties of execution contexts, it serves two primary purposes:

Author's Contact Information: Andrey Stoyan, HSE University, Saint Petersburg, Russian Federation, a.stoyan@hse.ru.

(1) An effect system makes effects explicit in type signatures, allowing functional abstraction boundaries to be specified with greater precision: implementation details are less likely to leak implicitly via context interactions. For example, a function cannot have an unspecified observable behavior such as mutation of external state.
(2) Furthermore, an effect system provides static enforcement of *effect safety* for a computation's use by ensuring that computations are only executed within appropriate contexts. So, an effect system must ensure that all exceptions will be handled and all functionality required by inversion of control is available.

It is also worth mentioning that effect systems can capture additional properties of computations, such as potential for divergence.

Many promising designs of effect systems were proposed: row-polymorphic types [13], capabilities [3, 4], modal types [30], etc. However, each of these approaches exhibits its own advantages and drawbacks, and the overall configuration of the design space for effect systems remains underexplored. At the same time we believe it is important for language designers to have a comprehensive view of the available design options, allowing them to select the most appropriate solution given the specific characteristics and constraints of their target language.

In this paper, we argue that effect systems can be conceptualized as a combination of two simpler and more foundational language features: implicit parameters and type-based escape analysis. Viewing effect systems from this perspective introduces axes within the effect system design space: varying the designs of these two features yields different possible instantiations. Moreover, this approach facilitates the seamless integration of effect systems into existing mainstream languages, as many of these languages already support at least one of these features, since each of which is independently practical.

We base our work on prior research, predominantly arising from the Scala language [3, 21]. Our primary contribution lies in articulating the idea of effect system decomposition explicitly and leveraging these insights to design a new effect system that is both practical and minimalistic.

Contributions of this paper are summarized as follows:

- We propose a perspective on effect system design as a combination of two properly-designed language features: implicit parameters and type-based escape analysis; and motivate our approach by looking at effect systems as systems for managing free variables (Section 2).
- We describe required properties of implicit parameters design and provide a suitable one using a formal calculi (Section 4).
- We provide a novel type-based escape analysis technique that only requires straight-forward dataflow descriptions from a programmer (Section 5).
- We discuss multiple approaches aimed at improving the usability of an effect system, making it more accessible and easier for broad adoption (Section 6).
- We compare our approach with previous art and show that other designs can be considered in implicit parameters and escape analysis basis (Section 7).

2 Two pieces of an effect system design

In this section we introduce the central idea of this paper through practical examples. Also, we illustrate that the pragmatic motivations for effect systems emerge naturally from common programming scenarios.

2.1 The road to implicit parameters

Consider our running example, where the business logic function executes an update operation on the database storage. The first version establishes its own database connection and executes a query:

```
fun business1() { let db = Db.open(...); db.query("...") }
```

This approach has several drawbacks. Firstly, it hinders testability, as it is not straightforward to replace the database connection with a mock or alternative implementation; consequently, verifying the business logic in isolation becomes challenging. Secondly, this function produces observable side effects — such as performing database operations — which are an integral part of its external contract. Nevertheless, these effects are not reflected in the function's signature.

The second version provides a notable improvement:

```
fun business2(db: Db) { db.query("...") }
fun caller() { let db = Db.open(...); business2(db) }
```

Here, the business logic function is parameterized over the database connection, enabling the call site to supply a concrete implementation with the desired semantics. Furthermore, the observable effects are now made explicit through the db parameter. However, this approach incurs the overhead of additional syntactic complexity at the call site, as the administrative effort of explicitly passing parameters increases. While this may appear trivial for a single argument, the burden becomes significant if, for example, four such parameters must be threaded through multiple layers of the call stack. Best practices intended for widespread adoption should be at least as accessible and convenient as less optimal alternatives.

To reduce boilerplate, the third version replaces ordinary parameter with dynamically scoped free variable `db`. Such variables acquire their values through lookup within the dynamic scope of the function call.

```
fun business3() { db.query("...") }
fun caller() { let db = Db.open(...); business3() }
```

However, this approach comes at the expense of other benefits: specifically, it sacrifices static typing and the explicit representation of context dependencies.

To regain static type safety, we approximate dynamically scoped free variables using implicit function parameters [15]. At the declaration site, we introduce a **`context`** keyword to define a group of implicit parameters, which are automatically supplied by the compiler at the call site. To make a value available for implicit parameter resolution, we use the **`context let`** declaration syntax.

```
context(db: Db)
fun business4() { db.query("...") }

fun caller() {
    context let db = Db.open(...)
    business4()
}
```

In terms of effects and contexts, **`context let`** introduces an execution context, whose scope is restricted to the corresponding lexical region, and provides functionality of access to a database storage. Implicit parameters express a function's requirements for its context. In order to invoke such a function, the context must establish its validity by supplying concrete values for these implicit parameters. We refer to these values as *capabilities*, and logically they serve as witnesses of context validity. It is important to note, that until this point our discussion was based solely on

the mechanism of bindings, and did not require any deep understanding of the abstract notion of effects.

Exceptions and other algebraic operations can be tracked in an analogous manner [21]. For example, a `throw SomeException()` construct can be interpreted as calling a function that requires an implicit parameter of type `Handler<SomeException>`. Correspondingly, constructs that handle exceptions must supply appropriate witness values for these implicit parameters.

```
fun checkingExceptions() {
    try { // provides a capability of type Handler<SomeException>
        throw SomeException() // requires a capability of type Handler<SomeException>
    } catch (e: SomeException) { ... }
}
```

Nowadays, several widely used programming languages provide support for implicit parameters, and the inference mechanisms associated with these parameters is well understood [15, 22, 28]. Moreover, implicit parameters alone are sufficient as the evidence-passing technique to implement tail-resumptive algebraic operations [33].

2.2 Retain effect safety with escape analysis

An effect system should ensure that all computations are executed within appropriate contexts — a property we refer to as *effect safety*. In our setting, the context must supply capabilities as evidence of its validity. Consequently, these capabilities must not be allowed to escape their corresponding contexts. For example, if a `Handler` value were to escape a corresponding `try-catch` block, we would lose the guarantee that all exceptions are handled and, thereby, the effect safety itself:

```
fun unsafe() {
    let f =
        try { // context let h: Handler<SomeException>
            let g: () -> Nothing = () => throw SomeException() // captures h
            g // leak computation capturing capability
        } catch (e: SomeException) { ... }
    f() // throws SomeException
}
```

To prevent the leakage of capabilities, some form of escape analysis is required [23]. Given that effect systems operate at the type level, it is natural to employ type-based techniques for escape analysis as well [34]. Furthermore, since this analysis must account for the flow of capabilities across function boundaries, it should be directly reflected in function signatures to be intra-procedural. In the following, we provide an informal overview of the type-based escape analysis technique introduced in this paper.

We refer to values that must not escape a given scope as *tracked*, and likewise, their types are designated as tracked. For instance, capabilities are tracked, as are, by default, the types of implicit function parameters. To differentiate between tracked and non-tracked types, each type `T` is annotated with a special label `lab`, referred to as a lifetime:

```
T'lab /* or for function types */ (A)'lab -> B
```

The `free` lifetime label designates non-tracked values. In contrast, the `local` label appears in tracked types and is treated specially by the type system to prevent their escape. For instance, the return type of the `try-catch` block must not include the `local` label. Consequently, the following code will be rejected by the type system, as the function's lifetime is determined by the lifetimes of the captured values:

```
fun safe() {
    let f =
        try { // context let h: Handler<SomeException>'local
            let g: ()'local -> Nothing = () => throw SomeException()
            g // type error: leaking type includes local
        } catch (e: SomeException) { ... }
    f()
}
```

Tracked values also cannot be returned from functions. Thus, we can specify in type of a higher-order function that it do not leak an argument function. For example, the eager `map` function calls its argument inplace:[1]

```
fun map(xs: List<a>, f: (a)'local -> b): List<b>
```

Note that it is safe to pass functions that capture capabilities to `map`, and this safety can be verified statically:

```
context(io: IO'local)
fun printAll(xs: List<Int>) {
    map(xs, (x) => io.print(x)) // capability capturing is safe here
}
```

In the example, the `io` is the lexically scoped free variable for the lambda function. The ability to close over capabilities enables the technique known as *contextual polymorphism* [4, 5]. This approach allows higher-order functions to remain fully transparent with respect to effects, eliminating the need for explicit polymorphic type variables that range over effect rows. In comparison, the corresponding function type in the Koka language employs row-polymorphic types, where the `e` denotes a type variable ranging over effect row types [13]:

```
fun map(xs: list<a>, f: (a) -> e b): e list<b>
```

To enable a function to propagate its arguments through its result, polymorphism over lifetime labels can be used. In essence, this is an explicit specification of the function's dataflow directly in the type signature. For instance, the lazy `map` function does not eagerly compute and return the final result; instead, it immediately yields a collection that encapsulates the processing logic. This behavior can be captured by assigning the same lifetime variable `l` to both the argument function `f` and the result type. Consequently, the type system records that any tracked capabilities captured by `f` may be retained within the result. Thus, such a function continues to support contextual polymorphism, as it can only permit `f` to escape with the result, and the call site is made aware that the result must not escape its prescribed context:

```
fun lazyMap(xs: LazyList<a>, f: (a)'l -> b): LazyList<b>'l

context(io: IO'local)
fun printAll(xs: LazyList<Int>) {
    let ys: LazyList<Int>'local = lazyMap(xs, (x) => io.print(x))
    ys.collect() // force execution
}
```

To express the capture of two or more lifetime-polymorphic values, we employ a syntax with plus that denotes the minimum of their lifetimes:

[1] We adopt the Haskell naming convention, where type names beginning with a lowercase letter denote type variables, which are implicitly universally quantified.

```
fun compose(f: (b)'l1 -> c, g: (a)'l2 -> b): (a)'l1+l2 -> c =
    (x) => f(g(x))
```

For the sake of simplicity, lifetime polymorphism is not supported for compound datatypes. Instead, their lifetimes are computed as the minimum of the lifetimes of their components; the same mechanism is applied to closures. Upon destructuring, the lifetimes of the individual components are, in turn, approximated by the lifetime of the original compound datatype.

```
fun makeRepository(file: File'l1, conn: Connection'l2): Repository'l1+l2 =
    Repository(file, conn)

context(repo: Repository'local)
fun registerUser(user: User) {
    let conn: Connection'local = repo.conn
    conn.send(user)
}
```

We define subtyping so that non-tracked values may be used in positions where tracked values are expected: `free <: local`. This design increases the number of safe well-typed programs and facilitates gradual adoption of effect tracking for languages transitioning from untyped effects to the effect system.

We permit tracked types to instantiate polymorphic type parameters, thereby making them first-class citizens in the language. To control when tracked types are appropriate for instantiation, we use bounded polymorphism. For example, the `map` function uses values of type `a` inside and returns values of type `b` directly to the caller. As a result, these type parameters may be instantiated with tracked types:[2] [3]

```
fun map(xs: List<a>, f: (a)'l -> b): List<b>'l where a <: Any'local, b <: Any'local
```

Upon capturing, an unknown lifetime of a type argument is approximated by the lifetime of its bound. Consider the most permissive type for the `lazyMap` function. For the parameter `a` we use the polymorphic bound `la` and include it into the resulting lifetime, since the resulting lazy list captures the initial list. The `ba` is not used in the result type, because it includes `b` itself.

```
fun lazyMap(xs: LazyList<a>, f: (a)'lf -> b): LazyList<b>'lf+la
        where a <: Any'la, b <: Any'lb
```

Various techniques can be employed to mitigate the boilerplate associated with lifetime polymorphism. For instance, the lifetime elision mechanism in the Rust language [20] utilizes heuristics to automatically insert lifetime variables, thus reducing the need for explicit annotations. This technique, along with dependent typing-like syntactic sugar, will be discussed in Section 6.1.

```
fun lazyMap(xs: LazyList<a>, f: (a)'lf -> b): LazyList<b>'f+xs
```

The utility of type-based escape analysis extends well beyond its role in the construction of effect systems. Specifically, it can be leveraged to reduce the number of heap allocations [18], enable borrowing for the ownership-based resource management [18, 20], and ensure the safety of non-local returns [1].

[2] We assume that `Any` serves as the top element in the type lattice.
[3] To preserve backward compatibility, the type `Any'free` may be used as the default bound.

Free variables	dynamically scoped	lexically scoped
Represent	unfulfilled contextual requirements	fulfilled contextual requirements
Dealt by	implicit parameters	type-based escape analysis
User specifies	more function parameters	a function's dataflow

Fig. 1. Effect systems are for managing free variables.

2.3 Summary

An effect system can be regarded as the integration of implicit parameters with type-based escape analysis. Consequently, the design of an effect system naturally decomposes into two distinct components and can be developed incrementally in a step-by-step manner.

In fact, our previous discussion has primarily focused on free variables. Indeed, a term inherently depends on its context via its free variables, acquiring different semantics depending on the particular context in which it is evaluated or defined. So, free variables are the target for effect systems, while bound once are regulated by canonical type systems.

Our perspective on effect systems arises from a fundamental observation: the semantics of free variables can be defined by two principal scoping disciplines — dynamic and lexical scoping. Adopting one or the other proves advantageous depending on the programming scenario. We summarize the correspondence between these two disciplines in Figure 1 and further discuss the relationship to existing approaches in effect system design in Section 7.

3 Core calculus

In this section, we introduce an explicitly typed, call-by-value core calculus, denoted as *Core*, which serves as the foundational framework for our study. In subsequent sections, we will extend *Core* with inference of implicits and type-based escape analysis.

3.1 Effect handlers

In our formal developments, we adopt effect handlers as a universal mechanism for defining effects [25, 26], since they are sufficiently expressive to capture all effects of interest. This obviates the need to treat individual effects separately. Furthermore, in contrast to monad transformers [16, 27], the dynamic semantics of effect handlers can be defined independently of any type-directed translation. Consequently, this approach imposes no inherent constraints on the design of the effect system, providing us greater flexibility.

We begin by providing a brief introduction to effect handlers and demonstrate how they can be used to implement the effects mentioned in the previous Section 2.

The `handle` construct allows to define an execution context for a particular scope by specifying a set of *effect operations*. Within this scope, the `perform` construct allows code to invoke these operations, facilitating interaction with the execution context managed by the handler. The implementation of each operation within a handler is granted access to a delimited continuation of the call site [6]. This continuation can be invoked to resume the computation with an operation's result.

In our setting, we employ lexically scoped handlers, which require that each `perform` invocation is associated with a specific handler instance [2, 5].

For example, consider defining an execution context that supplies a specific constant value to a computation analogous to `context let` from Section 2. This can be achieved by providing a handler for a `Reader` effect, which offers an `ask` operation that returns the designated constant. As in Koka, the `resume` identifier is used to refer to the corresponding delimited continuation. The following

computation sums the results of two invocations of `ask`; with implementation returning the constant 21, the overall result of the computation will be 42:

```
effect Reader<e> { op ask(): e }

handle r: Reader<Int> { op ask() resume(42) } // like 'context let r = 42'
perform r.ask() + perform r.ask()             // like 'r + r'
```

To implement exception handling, the `throw` operation can be defined to discard the current continuation, effectively aborting the remainder of the computation. For instance, the following program returns the first thrown value 1:

```
effect Exception<e> { op raise(e): a }

handle h: Exception<Int> { op raise(e) e } // like 'catch'
perform h.raise(1) + perform h.raise(2)    // like 'throw 1 + throw 2'
```

Other algebraic effects, such as mutable state and nondeterminism, can likewise be implemented using the `handle` construct [26].

3.2 Syntax of *Core*

Let us introduce the syntax of *Core*, as illustrated in Figure 2. We use an overline to denote a sequence of corresponding entries. Constructs highlighted in gray are included solely to support the operational semantics.

For explanatory purposes, we will continue to employ the informally defined surface syntax introduced in Section 2; its correspondence to the formal definitions is straightforward.

The main considerations that have shaped this syntax are as follows:

- The functional type $ctx\ (e)\ \overline{\tau} \to \sigma$ ensures that implicit parameters are always followed by ordinary parameters to prevent computations from being prematurely triggered by the inference of implicits (the overline denotes a tuple of argument types, with the empty tuple interpreted as the unit type).
- Corresponding to the functional type, at the term level we partition function arguments into two categories: *contextual arguments* (or implicit arguments), which are intended to be inferred automatically in future extensions, and ordinary arguments, which must be provided explicitly by the programmer.
- Capability constructors K_{cap} are used in an operational semantics to represent a capability.
- The *handle* construct introduces a capability associated with the newly created handler instance and binds it to a name. The `perform` construct, in turn, requires a capability in order to perform an effect operation. This approach helps us to naturally build an effect system from the perspective discussed in Section 2.
- For conciseness, we omit `return` blocks in handlers, since they do not contribute additional expressiveness.
- The $handler_m$ construct acts as a continuation delimiter, identified by a unique marker m for operational semantics purposes.
- The typing context differentiates regular bindings from *contextual bindings*, the latter of which are restricted to monotypes. While separate typing contexts for each binding class would be possible, this would necessitate different variable domains for contextual and ordinary bindings.

Note that the type of first-class functions contains a list of contextual argument types. This enables us to type abstractions over handlers. For example, the following function establishes an execution context by introducing a handler for an exception type `E`:

Variables		x, y, z, f, g, op
Values	v	$::= x \mid K \mid \Lambda\overline{\alpha}.\, v$
		$\mid K\ \overline{\tau}\ \overline{v} \mid$ $K_{cap}\ \overline{\tau}\ m\ h$
		$\mid \lambda\overline{x : \tau}\ \overline{y : \sigma}.\, t$
Terms	t, u	$::= v \mid t\ \overline{\tau} \mid t\ \overline{t}\ \overline{t}$
		$\mid match\ t\ \{\overline{K\ \overline{x} \to t}\}$
		$\mid perform\ op\ x\ \overline{\tau}\ \overline{t}$
		$\mid handle\ x : T\ \overline{\tau}\ \{h\}\ in\ t$
		$\mid$ $handler_m\ t$
Handlers	h	$::= \overline{op = t}$
Programs	p	$::= \epsilon \mid x : s = t, p$

Type variables		α, β
Type constructors		T
Monotypes	τ, σ, η	$::= \alpha \mid T\ \overline{\tau} \mid ctx\ (e)\ \ \overline{\tau} \to \sigma$
Type schemas	s	$::= \forall\overline{\alpha}.\, \tau$
Effect rows	e	$::= \epsilon \mid \tau, e$
Typing contexts	Γ, G	$::= \epsilon \mid x : s, \Gamma \mid x :_c \tau, \Gamma$
		$\mid K : \forall\overline{\alpha}.\, \overline{\tau} \to T\ \overline{\alpha}, \Gamma$
Effect contexts	Σ	$::= \{\overline{K_{cap} : \forall\overline{\alpha}.\, sig \to T\ \overline{\alpha}}\}$
Effect signatures	sig	$::= \{\overline{op : \forall\overline{\alpha}.\, \overline{\tau} \to \sigma}\}$
Free type variables		$ftv(\cdot)$

Evaluation context $E ::= \square \mid E\ \overline{\tau} \mid E\ \overline{t}\ \overline{t} \mid v\ (\overline{v}, E, \overline{t})\ \overline{t} \mid v\ \overline{v}\ (\overline{v}, E, \overline{t}) \mid K\ \overline{\tau}\ (\overline{v}, E, \overline{t})$
$\mid match\ E\ \{\overline{K\ \overline{x} \to t}\} \mid perform\ op\ x\ \overline{\tau}\ (\overline{v}, E, \overline{t}) \mid handler_m\ E$

Fig. 2. Syntax for *Core*.

```
fun withE(f: context(Handler<E>) () -> r): r =
    try { f() } catch (e: E) { ... }
```

We assume that the effect context Σ is populated by user-defined `effect` declarations, analogous to those found in Koka [14]. Specifically, for a program containing the following declaration,

```
effect Eff<a> {
    op id(x: b): b
    op yield(x: a): Unit
}
```

we have

$$\Sigma = \{Eff K_{cap} : \forall\alpha.\, \{id : \forall\beta.\, b \to b, yield : a \to Unit\} \to Eff\ a\}$$

There exists a bijection between capability constructors and effect types.

3.3 Operational semantics of *Core*

Let us highlight the most significant aspects of our operational semantics, as presented in Figure 3:

- **(handle)** rule selects the capability constructor associated with the specified effect type T and uses it to create a new capability. Additionally, it allocates a fresh marker m and installs it with the *handler* construct.
- **(perform)** rule deconstructs the provided capability, identifies the targeted operation, and invokes it with the current continuation, which is delimited by the associated marker m. Since we adopt deep handlers [9], the handler is reinstalled within the continuation.

It is important to note that every `perform` operation in our system is explicitly directed to a particular handler instance. As a result, there is no issue of accidental effect handling, so the effect encapsulation is achieved by default [5, 17, 37].

For instance, consider a function `withE1` that installs a handler for exception `E1`. If a computation `f` potentially raises exception `E2`, `withE1` will not be able to intercept this exception, since it does not provide `f` with the corresponding capability for `E2`:

```
fun withE1(f: context(Handler<E1>) () -> r): r =
    try { f() } catch (e: E1) { ... } catch (e: E2) { /* unreachable */ }
```

$$
\begin{array}{lllll}
\text{(tapp)} & (\Lambda\overline{\alpha}.\, t)\ \overline{\tau} & \longrightarrow & [\overline{\alpha \to \tau}]\ t & \\
\text{(app)} & (\lambda\overline{x}\ \overline{y}.\, t)\ \overline{v}\ \overline{v'} & \longrightarrow & [\overline{x \to v}, \overline{y \to v'}]\ t & \\
\text{(match)} & match\ K\ \overline{v}\ \{\overline{K_i\ \overline{x} \to u_i}\} & \longrightarrow & [\overline{x \to v}]\ u_k, \text{ where } K = K_k & \\
\text{(handle)} & handle\ x : T\ \overline{\tau}\ \{h\}\ in\ t & \longrightarrow & handler_m\ [x \to K_{cap}\ \overline{\tau}\ m\ h]\ t & \\
 & & \text{where} & m \text{ fresh}, (K_{cap} : \forall\overline{\alpha}.\, sig \to T\ \overline{\alpha}) \in \Sigma & \\
\text{(return)} & handler_m\ v & \longrightarrow & v & \\
\text{(perform)} & handler_m\ E[perform\ op\ (K_{cap}\ \overline{\tau_1}\ m\ h)\ \overline{\tau_2}\ \overline{v}] & \longrightarrow & t\ \overline{\tau_2}\ \overline{v}\ k & \\
 & & \text{where} & (op = t) \in h, \theta = [\overline{\alpha \to \tau_1}, \overline{\beta \to \tau_2}], & \\
 & & & (K_{cap} : \forall\overline{\alpha}.\, sig \to T\ \overline{\alpha}) \in \Sigma, & \\
 & \dfrac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}\ \text{(step)} & & (op : \forall\overline{\beta}.\, \overline{\sigma} \to \sigma') \in sig, & \\
 & & & k = \lambda x : \theta\ \sigma'.\, handler_m\ E[x] &
\end{array}
$$

Fig. 3. Operational semantics for *Core*.

Since handlers are propagated explicitly to the operation call site, our approach eliminates the need for dynamic handler lookup and the collection of the continuation when the callee operation is tail-resumptive [33]. In a low-level implementation, such a `perform` construct can thus be compiled to a direct virtual call, further improving efficiency.

3.4 Typing rules for *Core*

- The typing rules for *Core* (see Figure 4) are presented in algorithmic form.
- For conciseness, we omit both type well-formedness conditions and the (invariant) effect context Σ.
- Each entry in the effect context Σ can be uniquely identified by its capability type; thus, we treat Σ as a mapping: $\Sigma(T) = \langle\overline{\alpha}, sig\rangle \iff K_{cap} : \forall\overline{\alpha}.\, sig \to T\ \overline{\alpha} \in \Sigma$.
- All variable bindings within the typing context are assumed to be unique with respect to variable names, so rules Var and $CVar$ do not contradict each other.

4 Design of implicit parameters

In this section we discuss design decisions for implicit parameters feature that enhance its suitability as a component of a practical effect system. Throughout the discussion, we introduce a formal calculus with implicit parameters, denoted as $Core^{im}$, which extends the *Core* calculus.

4.1 Syntax of $Core^{im}$

We extend the syntax of *Core* by introducing *omitting application*, which allows the first block of contextual arguments to be unspecified:

$$\text{Terms} \quad t, u \quad ::= \ldots \mid t\ \overline{\ t}\ \mid \ldots$$

4.2 Inference of implicit parameters

We define a type-directed translation from $Core^{im}$ to *Core*, as presented in Figure 5. Deriving translation rules for other constructs from the typing rules for *Core* is straightforward.

The relation $\Gamma \vdash_e \overline{x}$ selects from the typing context those variables that participate in inference with the specified effect types. When multiple candidates exist for a type, the leftmost one is selected. Since function arguments are uncurried and thus appear logically simultaneously in the context, we require contextual arguments to have distinct types.

$$\boxed{\Gamma \vdash t : s}$$

$$\frac{x : s \in \Gamma}{\Gamma \vdash x : s}\ Var \qquad \frac{x :_c \tau \in \Gamma}{\Gamma \vdash x : \tau}\ CVar \qquad \frac{K : \forall \overline{\alpha}.\, \tau \to T\ \overline{\alpha} \in \Gamma}{\Gamma \vdash K : \forall \overline{\alpha}.\, \tau \to T\ \overline{\alpha}}\ Ctor$$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \Lambda \overline{\alpha}.\, v : \forall \overline{\alpha}.\, \tau}\ TLam \qquad \frac{\Gamma \vdash t : \forall \overline{\alpha}.\, \sigma}{\Gamma \vdash t\ \overline{\tau} : [\overline{\alpha \to \tau}]\ \sigma}\ TApp$$

$$\frac{\overline{x :_c \tau_1}, \overline{y : \tau_2}, \Gamma \vdash t : \sigma}{\Gamma \vdash \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\, t : ctx\ (\overline{\tau_1})\ \ \overline{\tau_2} \to \sigma}\ Lam \qquad \frac{\Gamma \vdash t : ctx\ (\overline{\tau_1})\ \ \overline{\tau_2} \to \sigma \quad \overline{\Gamma \vdash u_1 : \tau_1} \quad \overline{\Gamma \vdash u_2 : \tau_2}}{\Gamma \vdash t\ \overline{u_1}\ \overline{u_2} : \sigma}\ App$$

$$\frac{\Gamma \vdash t : T\ \overline{\tau} \quad \Gamma \vdash K_i : \forall \overline{\alpha}.\, \overline{\sigma_i} \to T\ \overline{\alpha} \quad \overline{x_i : [\overline{\alpha \to \tau}]\ \sigma_i}, \Gamma \vdash u_i : \eta}{\Gamma \vdash match\ t\ \{\overline{K_i\ \overline{x_i} \to u_i}\} : \eta}\ Match$$

$$\frac{\Gamma \vdash x : T\ \overline{\tau_1} \quad \Sigma(T) = \langle \overline{\alpha}, sig \rangle \quad op : \forall \overline{\beta}.\, \overline{\sigma} \to \eta \in sig \quad \theta = [\overline{\alpha \to \tau_1}, \overline{\beta \to \tau_2}] \quad \overline{\Gamma \vdash t : \theta\ \sigma}}{\Gamma \vdash perform\ op\ \overline{\tau_2}\ x\ \overline{t} : \theta\ \eta}\ Perform$$

$$\frac{x :_c T\ \overline{\tau}, \Gamma \vdash u : \eta \qquad \begin{array}{c} \Sigma(T) = \langle \overline{\alpha}, sig \rangle \qquad op_i : \forall \overline{\beta_i}.\, \overline{\sigma_i} \to \sigma_i' \in sig \\ \theta = [\overline{\alpha \to \tau}] \quad \Gamma \vdash \theta\ t_i : \forall \overline{\beta_i}.\, ctx\ ()\ \ (\overline{\theta\ \sigma_i}, \theta\ \sigma_i' \to \eta) \to \eta \end{array}}{\Gamma \vdash handle\ x : T\ \overline{\tau}\ \{\overline{op_i = t_i}\}\ in\ u : \eta}\ Handle$$

$$\frac{K_{cap} : \forall \overline{\alpha}.\, sig \to T\ \overline{\alpha} \in \Sigma}{\Gamma \vdash K_{cap}\ \overline{\tau}\ m\ h : T\ \overline{\tau}}\ Cap \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash handler_m\ t : \tau}\ Handler$$

$$\boxed{\Gamma \vdash p} \qquad \frac{}{\Gamma \vdash \epsilon}\ \epsilon Prog \qquad \frac{\Gamma \vdash t : s \quad ftv(s) = \emptyset \quad \Gamma \vdash p}{\Gamma \vdash x : s = t, p}\ Prog$$

Fig. 4. Typing rules for *Core*.

$$\boxed{\Gamma \vdash t : \tau \rightsquigarrow t_c}$$

$$\frac{\overline{x :_c \tau_1}, \overline{y : \tau_2}, \Gamma \vdash t : \sigma \rightsquigarrow t_c \quad distinct(\overline{\tau_1})}{\Gamma \vdash \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\, t : ctx\ (\overline{\tau_1})\ \ \overline{\tau_2} \to \sigma \rightsquigarrow \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\, t_c}\ Lam$$

$$\frac{\Gamma \vdash t : ctx\ (\overline{\tau_1})\ \ \overline{\tau_2} \to \sigma \rightsquigarrow t_c \quad \overline{\Gamma \vdash_{\tau_1} \overline{x}} \quad \overline{\Gamma \vdash u : \tau_2 \rightsquigarrow u_c}}{\Gamma \vdash t\ \overline{u} : \sigma \rightsquigarrow t_c\ \overline{x}\ \overline{u_c}}\ CtxApp$$

$$\boxed{\Gamma \vdash_\tau x} \qquad \frac{y :_c \tau \notin G}{G, x :_c \tau, \Gamma \vdash_\tau x}\ Infer$$

Fig. 5. Inference of implicit parameters.

Note that, since the typing context maintains only bindings with unique names (see Section 3.4), it is possible for a user to shadow a contextual binding with an ordinary one, causing it to be excluded from subsequent inference. This issue can be addressed, for example, by representing the typing context as a list that permits duplicating entries and by assigning additional fresh names to contextual bindings. Implicits inference should use these names to refer directly to contextual binding avoiding shadowing.

There is a design decision to be made, whether inference should depend solely on types, or additionally rely on variable names. The latter approach is more complex, as it necessitates storing

variable names in effect rows and, furthermore, precludes the renaming of implicit parameters under α-equivalence. This restriction can make programs more fragile with respect to renaming. On the other hand, this approach permits simultaneous use of multiple effects of the same type.

However, in this work, we use the first approach and argue that, in practical programming, relying on multiple effects of the same type may be undesirable. Instead, we advocate for the definition and use of domain-specific effects rather than directly employing general-purpose ones. For instance, a domain-specific effect such as `UserRepository` offers a higher level of abstraction compared to a general-purpose `Database` effect.

Another design decision concerns the timing of inference of implicit parameters: should it occur immediately upon variable mention, or should it be deferred until application? This question is analogous to type instantiation in the presence of first-class polymorphism, where type inference determines type parameters [7]. We consider that immediate substitution is preferable, because it is convenient when working with contextually polymorphic functions, such as `map(xs, f)` [28]. However, in our formal elaboration we use another approach for brevity.

Actually, this choice determines the default behavior, since either approach can be emulated by the other via η-expansion. For instance, let us assume that substitution is performed at application time:

```
context(_: IO) fun f(): Unit
fun g(_: () -> Unit): Unit
g(() => f()) // captures IO capability
```

Alternatively, assume that substitution occurs at the point where a function is mentioned:

```
context(_: IO) fun f(): Unit
fun g(_: context(IO) () -> Unit): Unit
g(context(_: IO) () => f()) // abstracts over contextual parameters
```

4.3 System $Core^{im+<:}$ with implicit parameters and subtyping on effect rows

It is desirable to support a form of subtyping on effect rows, as this allows functions with fewer contextual requirements to be used in more permissive contexts in a natural way. For example, `withStd` is a function that provides default implementation for several effects. We should therefore be able to pass to `withStd` a function that only utilizes a subset of those effects:

```
fun withStd(f: context(IO, Allocator, Random) () -> r): r
withStd(g) // g : context(IO) () -> Unit
```

Contexts can be modeled as structural record types with width and permutation subtyping. However, a straightforward implementation of such records as dictionaries tends to be inefficient in terms of both time and space complexity.

To enable efficient subtyping, we restrict subtyping to be shallow, meaning that one function type is a subtype of another only if their effect rows are in a subtyping relationship, while the argument and return types remain identical:

$$\frac{e' <: e}{ctx\ (e)\ \ \tau \to \sigma <:_c ctx\ (e')\ \ \tau \to \sigma}\ FunSub$$

With this restriction, shallow subtyping can be implemented via η-expansion (see Figure 6). The relation $\Gamma \vdash t \Leftarrow \tau \rightsquigarrow t_c$ applies η-expansion to t when a functional type τ is expected.

$$\frac{\boxed{\Gamma \vdash t : \tau \rightsquigarrow t_c} \quad \Gamma \vdash t : ctx\ (\overline{\tau_1})\ \ \overline{\tau_2} \to \sigma \rightsquigarrow t_c \quad \overline{\Gamma \vdash_{\tau_1} x} \quad \overline{\Gamma \vdash u : \tau_2'} \quad \overline{\tau_2' <:_c \tau_2} \quad \overline{\Gamma \vdash u \Leftarrow \tau_2 \rightsquigarrow u_c}}{\Gamma \vdash t\ \overline{u} : \sigma \rightsquigarrow t_c\ \overline{x}\ \overline{u_c}}\ App$$

$$\frac{\boxed{\Gamma \vdash t \Leftarrow \tau \rightsquigarrow t_c} \quad \Gamma \vdash t : \tau \rightsquigarrow t_c}{\Gamma \vdash t \Leftarrow \tau \rightsquigarrow t_c}\ AppArg$$

$$\frac{\Gamma \vdash t : ctx\ \left(\overline{\tau_1'}\right)\ \ \overline{\tau_2} \to \sigma \rightsquigarrow t_c \quad \overline{\overline{x :_c \tau_1} \vdash_{\tau_2'} x'} \quad \overline{x\ \text{fresh}} \quad \overline{y\ \text{fresh}}}{\Gamma \vdash t \Leftarrow ctx\ (\overline{\tau_1})\ \ \overline{\tau_2} \to \sigma \rightsquigarrow \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\ t_c\ \overline{x'}\ \overline{y}}\ AppArgCtx$$

Fig. 6. Application rule for $Core^{im+<:}$.

5 Design of type-based escape analysis

In this section we develop an extension to the previously discussed systems that statically guarantees that no capabilities escape their corresponding handlers. The core aspects of this design have already been introduced in Section 2.2.

5.1 Syntax of $Core_\Delta$

Let us introduce $Core_\Delta$, an extension of *Core* that incorporates lifetimes, lifetime polymorphism, and bounded polymorphism. Modifications to the type syntax are presented in Figure 7, with changes highlighted in gray.

- Monotypes are now annotated with explicit lifetime labels. The syntax is deliberately chosen to be identical to that of type arguments to emphasize that lifetimes can be regarded as ordinary types, albeit of a distinct kind.
- Type schemas support abstraction over lifetime variables and permit the specification of polymorphic bounds.
- The resulting lifetime associated with data constructors is computed as the minimum of all lifetimes corresponding to those values that may later be extracted — that is, lifetimes encountered in positive positions.
- For brevity, we assume *Any local* as the default constraint for data constructor generics and *Any free* for capability constructors and effect operations.

Let us explicitly define some operators. The definitions of others are straight-forward. The $lt^+_\Gamma(\cdot)$ is defined as follows:

$$\begin{array}{lcl} lt^+_\Gamma(\alpha) & = & lt^+_\Gamma(\tau) \text{ if } \alpha <: \tau \in \Gamma, \emptyset \text{ otherwise} \\ lt^+_\Gamma(T\ \Delta\ \overline{\tau}) & = & \overline{lt^+_\Gamma(\tau)} \cup \overline{lt^-_\Gamma(\tau)} \cup \{\Delta\} \\ lt^-_\Gamma(T\ \Delta\ \overline{\tau}) & = & \overline{lt^+_\Gamma(\tau)} \cup \overline{lt^-_\Gamma(\tau)} \\ lt^+_\Gamma(ctx\ (\overline{\tau})\ \ \Delta\ \overline{\sigma} \to \overline{\eta}) & = & \overline{lt^-_\Gamma(\tau)} \cup \overline{lt^-_\Gamma(\sigma)} \cup \overline{lt^+_\Gamma(\eta)} \cup \{\Delta\} \\ lt^-_\Gamma(ctx\ (\overline{\tau})\ \ \Delta\ \overline{\sigma} \to \overline{\eta}) & = & \overline{lt^+_\Gamma(\tau)} \cup \overline{lt^+_\Gamma(\sigma)} \cup \overline{lt^-_\Gamma(\eta)} \end{array}$$

The lifetime of a type variable is approximated by the lifetime of its bounding type, if such a bound exists. In some cases, we intentionally employ an empty context to exclude the influence of bounds. Also, since type constructor parameters are invariant, they are treated as both positive and negative simultaneously.

Type variables		α, β
Type constructors		T
Lifetime variables		l
Lifetimes	Δ	$::= l \mid local \mid free \mid +\overline{\Delta} \mid \star$
Monotypes	τ, σ, η	$::= \alpha \mid T\ \Delta\ \overline{\tau} \mid ctx\,(e)\ \Delta\ \overline{\tau} \to \sigma$
Type schemas	s	$::= \forall\ \overline{l}\ \overline{(\alpha <: \tau)}.\ \sigma$
Position sign	p	$::= + \mid - \mid inv$
Free type variables		$ftv^p(\cdot)$
Lifetimes		$lt^p_\Gamma(\cdot)$
Free lifetime variables		$flt^p(\cdot)$
Least upper bound		$lub(\cdot)$
Eliminate lifetimes		$elim^p_{\Delta,\bar{l}}(\cdot)$
Effect rows	e	$::= \epsilon \mid \tau, e$
Typing contexts	Γ	$::= \emptyset \mid x : s, \Gamma \mid x :_c \tau, \Gamma$
		$\mid\ \overline{l <: \Delta, \Gamma} \mid \overline{\alpha <: \tau, \Gamma}$
		$\mid K : \forall\ \overline{l}\ \overline{\alpha}.\overline{\tau} \to T\ \overline{\left(+\overline{lt^+_\emptyset(\tau)}\right)}\ \overline{\alpha}, \Gamma$
Effect contexts	Σ	$::= \{K_{cap} : \forall\overline{\alpha}.\ sig \to T\ local\ \overline{\alpha}\}$
Effect signatures	sig	$::= \{\overline{op : \forall\overline{\alpha}.\overline{\tau} \to \sigma}\}$

Fig. 7. Syntax of types for $Core_\Delta$.

The lifetime elimination operator $elim^p_{\Delta,\bar{l}}(\cdot)$ approximates the designated existential lifetimes $\bar{l}$ to prevent their leakage outside the scope of the pattern-matching branch. It is defined as following:

$$
\begin{aligned}
elim^p_{\Delta,\bar{l}}(\alpha) &= \alpha \\
elim^p_{\Delta,\bar{l}}(K\ \Delta\ \overline{\tau}) &= K\ \left(elim^p_{\Delta,\bar{l}}(\Delta)\right)\ \left(\overline{elim^{inv}_{\Delta,\bar{l}}(\tau)}\right) \\
elim^p_{\Delta,\bar{l}}(ctx\,(\overline{\tau_1})\ \Delta\ \overline{\tau_2} \to \sigma) &= ctx\ \left(\overline{elim^{p^{-1}}_{\Delta,\bar{l}}(\tau_1)}\right)\ \left(elim^p_{\Delta,\bar{l}}(\Delta)\right)\ \left(\overline{elim^{p^{-1}}_{\Delta,\bar{l}}(\tau_2)}\right) \to elim^p_{\Delta,\bar{l}}(\sigma) \\
elim^p_{\Delta,\bar{l}}(l') &= l',\ \text{if } l' \notin \bar{l} \quad \text{else} \quad \Delta,\ \text{if } p = + \quad \text{else} \\
&\quad\ free,\ \text{if } p = - \quad \text{else} \quad \star,\ \text{if } p = inv \\
elim^p_{\Delta,\bar{l}}(+\overline{\Delta'}) &= +\left(\overline{elim^p_{\Delta,\bar{l}}(\Delta')}\right)
\end{aligned}
$$

When the position is invariant and it is not possible to approximate a lifetime by a concrete bound, we employ the special lifetime $\star$, which denotes an unknown lifetime. We consider types with $\star$ lifetime tracked as well.

Syntax of terms follows the changes in the syntax of types:

Values	v	$::= \ldots \mid \Lambda\ \overline{l}\ \overline{\alpha}.\,v \mid K\ \overline{\Delta}\ \overline{\tau}\ \overline{v} \mid \ldots$
Terms	t, u	$::= \ldots \mid t\ \overline{\Delta}\ \overline{\tau} \mid handle\ x : T\ local\ \overline{\tau}\ with\ h\ in\ t \mid \ldots$

5.2 Typing rules for $Core_{\Delta}$

Let us explain several key aspects of the typing rules for $Core_{\Delta}$ (Figure 8):

- **Lam:** The typing rule for λ-abstractions guarantees that no tracked value can escape the scope of the function via its return value. Analogous to the treatment of data constructors, the lifetime associated with a function type is determined as the minimum of the lifetimes of all captured variables, excluding those that are present within the function type by themselves. This ensures that the function itself cannot outlive any tracked capabilities it closes over.
- **Match:** Existential lifetimes of a data constructor are treated as covariant projections bounded by the lifetime of the type. So, we substitute fresh lifetimes l' in their place. Furthermore, these fresh lifetime variables are eliminated from the resulting type of the branch.
- **Handle:** The *Handle* rule plays crucial role in ensuring effect safety. First, it annotates capabilities created within the handler with the lifetime label *local* and prohibits occurrences of *local* and $\star$ in positive positions within the result type η. This restriction prevents capabilities from escaping the handler's scope via returned values. Second, the rule enforces a well-formedness condition on effect signatures to ensure that no capabilities can be passed passed as arguments to operation calls. Without this constraint, a capability introduced by an inner handler could be propagated to an operation in an outer handler, thus escaping its scope. This restriction can be relaxed under certain conditions; further discussion is provided in Section 5.5.
- **TypesOf:** This rule extracts the types of designated variables from the typing context without discriminating between ordinary bindings and contextual bindings.

5.3 Lifetime tunnelling

Our system allows tracked types to instantiate type parameters. Following the terminology introduced by Boruch-Gruszecki et al. [3], we refer to this mechanism as *lifetime tunnelling*.

Consider, for example, a function that updates the first component of a pair. In our approach, it suffices to specify appropriate bounds for the type parameters to ensure that the function correctly operates on tracked types. Furthermore, note that if the function is instantiated with untracked types, its result will likewise remain untracked: no superfluous constraints will be imposed.

```
fun mapFirst(p: Pair<a, b>, f: (a) -> c): Pair<c, b>
        where a <: Any'local, b <: Any'local, c <: Any'local {
    Pair(f(fst(p)), snd(p))
}
```

When capturing, we approximate the lifetimes of type parameters using the lifetimes of their bounds. To improve precision, polymorphic bounds can be specified for lifetimes:

```
fun mapFirst(p: Pair<a, b>): ((a) -> c)'la+lb -> Pair<c, b>
        where a <: Any'la, b < Any'lb, c <: Any'lc {
    (f) => Pair(f(fst(p)), snd(p))
}
```

5.4 Integration with implicit parameters

To extend $Core_{\Delta}$ with implicit parameters, resulting in $Core_{\Delta}^{im}$, it is necessary to incorporate omitting application into both the syntax and the translation rules, as described in Section 4. However, several details require careful consideration.

$$\boxed{\Gamma \vdash t : s}$$

$$\frac{x : s \in \Gamma}{\Gamma \vdash x : s}\ Var \qquad \frac{x :_c \tau \in \Gamma}{\Gamma \vdash x : \tau}\ CVar \qquad \frac{K : \forall \overline{l}\ \overline{\alpha}.\ \overline{\tau} \to T\ \left(+\overline{lt^+_\emptyset(\tau)}\right)\ \overline{\alpha} \in \Gamma}{\Gamma \vdash K : \forall \overline{l}\ \overline{\alpha}.\ \overline{\tau} \to T\ \left(+\overline{lt^+_\emptyset(\tau)}\right)\ \overline{\alpha}}\ Ctor$$

$$\frac{\overline{\alpha <: \sigma}, \Gamma \vdash t : \tau}{\Gamma \vdash \Lambda \overline{l}\ \overline{(\alpha <: \sigma)}.\ t : \forall \overline{l}\ \overline{(\alpha <: \sigma)}.\ \tau}\ TLam \qquad \frac{x : \forall \overline{l}\ \overline{(\alpha <: \tau)}.\ \sigma \in \Gamma \quad \overline{\Gamma \vdash \tau' <: [\overline{l \to \Delta}]\ \tau}}{\Gamma \vdash x\ \overline{\Delta}\ \overline{\tau'} : [\overline{\alpha \to \tau'}]\ [\overline{l \to \Delta}]\ \sigma}\ TApp$$

$$\frac{\overline{x :_c \tau_1}, \overline{y : \tau_2}, \Gamma \vdash t : \sigma \quad \Gamma \vdash_{fv(t)\setminus\overline{x}\setminus\overline{y}} \overline{s} \quad local, \star \notin lt^+_\emptyset(\sigma)}{\Gamma \vdash \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\ t : ctx\ (\overline{\tau_1})\ \left(+lt^+_{\Gamma \setminus ftv^-(\overline{\tau_1} \cup \overline{\tau_2}) \setminus ftv^+(\sigma)}(\overline{s})\right)\ \overline{\tau_2} \to \sigma}\ Lam$$

$$\frac{\Gamma \vdash t : ctx\ (\overline{\tau_1})\ \Delta\ \overline{\tau_2} \to \sigma \qquad \frac{\overline{\Gamma \vdash u_1 : \tau'_1}}{\Gamma \vdash \tau'_1 <: \tau_1} \qquad \frac{\overline{\Gamma \vdash u_2 : \tau'_2}}{\Gamma \vdash \tau'_2 <: \tau_2}}{\Gamma \vdash t\ \overline{u_1}\ \overline{u_2} : \sigma}\ App$$

$$\frac{\Gamma \vdash t : T\ \Delta\ \overline{\tau} \qquad \Gamma \vdash K_i : \forall \overline{l_i}\ \overline{\alpha}.\ \overline{\sigma_i} \to T\ \left(+\overline{lt^+_\emptyset(\sigma_i)}\right)\ \overline{\alpha} \qquad \overline{l'_i\ \text{fresh}} \qquad \overline{l'_i <: \Delta, x_i : [\overline{\alpha \to \tau}]\ [\overline{l_i \to l'_i}]\ \sigma'_i}, \Gamma \vdash u_i : \eta_i}{\Gamma \vdash match\ t\ \{\overline{K_i\ \overline{x_i} \to u_i}\} : lub\left(\overline{elim^+_{\Delta, \overline{l'_i}}(\eta_i)}\right)}\ Match$$

$$\frac{\Sigma(T) = \langle \overline{\alpha}, sig \rangle \qquad \Gamma \vdash x : T\ \Delta\ \overline{\tau_1} \qquad \overline{\Gamma \vdash t : \sigma'} \\ op : \forall \overline{\beta}.\ \overline{\sigma} \to \eta \in sig \quad \theta = [\overline{\alpha \to \tau_1}, \overline{\beta \to \tau_2}] \qquad \overline{\Gamma \vdash \sigma' <: \theta\ \sigma}}{\Gamma \vdash perform\ op\ \overline{\tau_2}\ x\ \overline{t} : \theta\ \eta}\ Perform$$

$$\frac{\begin{array}{c}\Sigma(T) = \langle \overline{\alpha}, sig \rangle \qquad x :_c T\ local\ \overline{\tau}, \Gamma \vdash u : \eta \\ \theta = [\overline{\alpha \to \tau}] \qquad local, \star \notin lt^+_\emptyset(\eta) \cup \overline{lt^+_{\overline{\beta <: \hat{\sigma}}, \Gamma}(\theta\ \sigma_i)} \\ \hat{\sigma} = Any\ free \quad \Gamma \vdash \theta\ t_i : \forall \overline{(\beta <: \hat{\sigma})}.\ ctx\ ()\ \ (\overline{\theta\ \sigma_i}, \theta\ \sigma'_i \to \eta) \to \eta'_i \\ \Gamma \vdash \eta'_i <: \eta \qquad \overline{\beta} \cap ftv(\eta'_i) = \emptyset \quad op_i : \forall \overline{\beta_i}.\ \overline{\sigma_i} \to \sigma'_i \in sig\end{array}}{\Gamma \vdash handle\ x : T\ local\ \overline{\tau}\ \{\overline{op_i = t_i}\}\ in\ u : \eta}\ Handle$$

$$\frac{K_{cap} : \forall \overline{\alpha}.\ sig \to T\ local\ \overline{\alpha} \in \Sigma}{\Gamma \vdash K_{cap}\ \overline{\tau}\ m\ h : T\ local\ \overline{\tau}}\ Cap \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash handler_m\ t : \tau}\ Handler$$

$$\boxed{\Gamma \vdash_{\overline{x}} \overline{\tau}}\ \frac{\overline{x :. s} \subset \Gamma}{\Gamma \vdash_{\overline{x}} \overline{s}}\ TypesOf \qquad \boxed{\Gamma \vdash p}\ \frac{}{\Gamma \vdash \epsilon}\ \epsilon Prog \qquad \frac{\Gamma \vdash t : s \quad flt(s) \cup ftv(s) = \emptyset \quad \Gamma \vdash p}{\Gamma \vdash x : s = t, p}\ Prog$$

Fig. 8. Typing rules for $Core_\Delta$.

Previously, to select appropriate implicit arguments, we used type equality between the type of the contextual binding and the expected type of the implicit parameter (see Figure 5). However, since $Core_\Delta$ supports subtyping, we revise these rules as shown in Figure 10.

The $Infer$ rule selects the nearest appropriate binding that suits as an argument for an implicit parameter of type τ. Additionally, the $distinct_{<:}$ operator ensures that least upper bound of every pair of argument types results in Any type constructor.

To support subtyping on effect rows via η-expansion (Section 4.3), we introduce an additional subtyping relation specifically for the application rule. This relation treats functional types by

$$\boxed{\Gamma \vdash \sigma <: \tau}$$

$$\frac{\alpha <: \tau \in \Gamma}{\Gamma \vdash \alpha <: \tau}\ SubCtx \qquad \frac{\Gamma \vdash \Delta' <: \Delta}{\Gamma \vdash K'\ \Delta'\ \overline{\tau} <: K\ \Delta\ \overline{\tau}}\ SubData$$

$$\frac{\Gamma \vdash \Delta' + lt^{+}_{\Gamma}(\tau) <: \Delta}{\Gamma \vdash \tau <: Any\ \Delta}\ SubAny \qquad \frac{\overline{\Gamma \vdash \tau_1 <: \tau_1'} \quad \Gamma \vdash \Delta' <: \Delta \quad \overline{\Gamma \vdash \tau_2 <: \tau_2'} \quad \sigma' <: \sigma}{\Gamma \vdash ctx\left(\overline{\tau_1'}\right)\ \Delta'\ \overline{\tau_2'} \to \sigma' <: ctx\ (\overline{\tau_1})\ \Delta\ \overline{\tau_2} \to \sigma}\ SubFun$$

$$\boxed{\Gamma \vdash \Delta' <: \Delta}$$

$$\frac{}{\Gamma \vdash free <: \Delta}\ SubFree \qquad \frac{}{\Gamma \vdash \Delta <: local}\ SubLocal$$

$$\frac{l <: \Delta' \in \Gamma \quad \Gamma \vdash \Delta' <: \Delta}{\Gamma \vdash l <: \Delta}\ SubCtx_{\Delta} \qquad \frac{\forall \Delta' \in \overline{\Delta'}.\ \exists \Delta \in \overline{\Delta}.\ \Gamma \vdash \Delta' <: \Delta}{\Gamma \vdash +\overline{\Delta'} <: +\overline{\Delta}}\ Sub+$$

Fig. 9. Subtyping rules for $Core_{\Delta}$.

$$\boxed{\Gamma \vdash t : \tau \rightsquigarrow t_c}$$

$$\frac{\overline{x :_c \tau_1}, \overline{y : \tau_2}, \Gamma \vdash t : \sigma \rightsquigarrow t_c \quad distinct_{<:}(\overline{\tau_1})}{\Gamma \vdash \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\ t : ctx\ (\overline{\tau_1})\ \overline{\tau_2} \to \sigma \rightsquigarrow \lambda \overline{x : \tau_1}\ \overline{y : \tau_2}.\ t_c}\ Lam$$

$$\frac{\Gamma \vdash t : ctx\ (\overline{\tau_1})\ \overline{\tau_2} \to \sigma \rightsquigarrow t_c \quad \overline{\Gamma \vdash_{\tau_1} x} \quad \overline{\Gamma \vdash x : \tau_2} \quad \overline{\Gamma \vdash \tau_2 <: \tau_1} \quad \overline{\Gamma \vdash u : \tau \rightsquigarrow u_c}}{\Gamma \vdash t\ \overline{u} : \sigma \rightsquigarrow t_c\ \overline{x}\ \overline{u_c}}\ CtxApp$$

$$\boxed{\Gamma \vdash_{\tau} x} \qquad \frac{\Gamma \vdash \sigma <: \tau \quad \nexists \sigma'.\ \sigma' <: \sigma \wedge y :_c \sigma' \in G}{G, x :_c \sigma, \Gamma \vdash_{\tau} x}\ Infer$$

Fig. 10. Inference of implicit parameters for $Core^{im}_{\Delta}$.

verifying subtyping on effect rows as a whole, rather than performing pointwise subtyping on individual effect types:

$$\frac{\Gamma \vdash e <: e' \quad \overline{\Gamma \vdash \tau <: \tau'} \quad \Gamma \vdash \Delta' <: \Delta \quad \Gamma \vdash \eta' <: \eta}{\Gamma \vdash ctx\ (e')\ \Delta'\ (\overline{\tau'}) \to \sigma' <:_c ctx\ (e)\ \Delta\ (\overline{\tau}) \to \sigma}\ SubFunEff$$

5.5 Using effects with tracked values

Note that, at present, it is not possible to pass tracked values to effect operations. This limitation arises from the fact that capabilities are tracked, and their usage always requires an associated delimited continuation — the corresponding marker becomes invalid outside the scope of the relevant handler. However, if the use of a tracked value does not involve a continuation, it is safe to lend it to an operation call. Examples of such values include tracked resources or capabilities that witness tail-resumptive operations.

To enable this, we can introduce an additional lifetime, *regional*, with the ordering $free <: regional <: local$. This *regional* lifetime differs from *local* in two key aspects: it can bound generics of effect operations, and it can appear within the parameter types of operations.

In fact, permitting the passing of arbitrary capabilities to an operation effectively corresponds to representing higher-order effects [31, 32, 38]. We defer the investigation of this matter to future work.

6 Adoption in mainstream languages

The adoption of a substantial feature such as an effect system in a mainstream programming language presents a significant challenge. The design of the effect system should be straightforward and leverage existing developer expertise. Moreover, it must minimize boilerplate and ensure compatibility with widely used language features. Lastly, the effect system should support a clear pathway for gradual adoption, alongside providing escape mechanisms that allow complex scenarios to be addressed outside the effect system, with the option for subsequent integration.

In this section, we provide an informal discussion of potential extensions to our effect system design, aimed at improving it with respect to the aforementioned requirements.

6.1 Simplifying lifetime polymorphism

We consider the lifetime polymorphism the most problematic aspect of our design. A user should manually introduce lifetime names and link lifetimes of result with lifetimes of arguments in a relational manner:

```
fun compose
    <la, lb, lc, lf, lg, a <: Any'la, b <: Any'lb, c <: Any'lc>
    (f: (b)'lf -> c, g: (a)'lg -> b): (a)'lb+lf+lg -> c
```

One possible solution is to employ the lifetime elision mechanism [20], which automatically inserts lifetimes into type signatures based on heuristics. However, these heuristics may not always produce the intended results, necessitating manual lifetime annotations by the user in certain cases. In our example, this approach can look like the following (prime requires compiler to elide a lifetime):

```
fun compose<a', b', c'>(f: (b)' -> c, g: (a)' -> b): (a)' -> c
```

An alternative solution is to introduce special syntax in the surface language that enables direct specification of capturing through term variables. Although the syntax resembles dependent typing, we expect it to be fully implementable through a straightforward desugaring process that translates the surface language into the core calculus with lifetime polymorphism.

```
fun compose(f: (b) -> c, g: (a) -> b): (a)'f+g -> c
```

This solution may necessitate the introduction of explicit names for generic arguments in order to precisely specify more complex data flows:

```
fun makeRepository(files: Map<String, file: File>): Repository'file
```

In this case, it is crucial that the generated lifetime parameters remain always inferrable. We defer a thorough development of this surface syntax and its translation to future work.

6.2 Gradual adoption of the effect system

The effect system we present supports gradual adoption. We outline the stages of adoption in user code as follows:

(1) No effect system. All standard effects, such as I/O and exceptions, are available globally through untracked capabilities.
(2) Partial effect system. Capabilities remain untracked and are no longer globally available; instead, they must be explicitly passed via contextual parameters.
(3) Full effect system. Capabilities are tracked, and functions are required to specify their dataflows explicitly in signatures.

Furthermore, a practical language design should include an unsafe type cast to untracked types, enabling developers to handle complex scenarios without dealing escape analysis.

In our design, all effects are bound to specific handlers (see Section 3.3); that is, exceptions are aware of their handlers at the moment they are thrown. However, in practice, there are scenarios where code may need to violate effect encapsulation—for example, by intercepting all exceptions, storing them, and rethrowing later. Designing such a mechanism in a safe and sound manner presents significant challenges.

6.3 Working with OOP

Object-oriented programming remains a widely adopted paradigm in contemporary software development, presenting notable challenges for effect systems. Consider the following example: an interface declares a set of method signatures to which all implementations must conform. However, what happens if an implementation needs to throw exceptions beyond those specified in the interface's signature?

Note that only the code that is aware about the specific implementation type has enough information to properly handle implementation-specific exceptions. So, we can capture all needed capabilities on the object constructor call site:

```
interface I { fun f() }

class Impl : I {
    context(let e: Handler<SomeException>) init() {} // receive and store capability
    override fun f() { throw SomeException() } // uses stored capability
}

fun test(i: I'local) { i.f() }

fun main() {
    try {
        let i = Impl() // captures capabilities
        test(i)
    } catch (e: SomeException) { ... }
}
```

7 Related work

In this section we give a brief overview of existing solutions, consider them from our perspective on effect systems and provide a competitive analysis of our design. For a more comprehensive overview, see [29].

7.1 Row-polymorphic effect systems

Conventional row-based effect systems, first introduced by Lucassen and Gifford [19], represent the most classical approach to effect systems construction. Notable implementations of row-based effect systems include the Koka language [13, 14] and the Links language [8]. In addition, Haskell's `mtl` library[4] emulates row types by using type class constraints to represent lists of effects [10].

Such effect systems employ effect polymorphism to type higher-order functions, expressing that a function produces at least the same effects as its argument function. Although this approach is relatively simple and well-studied, it is often regarded as verbose and somewhat alien to conventional language design. It introduces a distinct concept of effects that programmers must consistently consider, which can increase cognitive overhead and reduce accessibility.

[4] https://hackage.haskell.org/package/mtl

Row-based effect systems can be conceptualized as systems that type dynamically bound free variables (i.e., effect operations) in an effect-style manner, analogous to the Reader monad. In contrast, our approach leverages implicit parameters, which correspond to a coeffect-style [24], and supports lexically scoped free variables as well.

7.2 Capability-based effect systems

In capability-based effect systems, instead of directly tracking the use of effect operations, the presence of special objects known as capabilities is monitored; such objects are required to perform effect operations. This approach has led to the realization that effect systems are fundamentally concerned with the management of free variables, and has enabled the discovery of contextual polymorphism [4, 5].

While implicit parameters generally do not pose significant challenges, the escape analysis of capabilities remains a complex problem. One approach employs higher-rank polymorphism to prevent capabilities from escaping [12, 34]. Another notable method is capture checking [3] in the Scala language, which facilitates dependent types for escape analysis; in this setting, variable names correspond to concrete lifetimes and the root capability aligns with our `local`. In contrast, our design does not utilize dependent typing and instead is based on well-understood bounded polymorphism and subtyping. Furthermore, we compute lifetimes of type variables by approximating them with the lifetimes of their bounds, whereas capture types employ more intricate boxing mechanisms, which remain unsettled at the time of writing [35, 36].

7.3 Modal effect systems

Modal effect systems [30, 39] use special modal type constructors to represent unfulfilled contextual requirements of a computation. When such a computation is placed within an appropriate context, it is automatically unboxed and becomes ready for execution.

From the perspective of tracking free variables, modal types as presented by Tang et al. [30] employ a kind system to distinguish between computations that may capture lexically scoped free variables (kind *Any*) and those that cannot (kind *Abs*). When a computation of kind *Any* is transferred between execution contexts, such as when it is returned from a handler, the type system augments its type with an additional modality. In other words, a process akin to the uncapturing of lexically scoped free variables is performed, whereby these variables become unfulfilled contextual requirements once again. Thus, this form of escape analysis facilitates control over leaking instead of strictly prohibiting it.

8 Conclusion

In this paper, we have explicitly articulated a conceptual framework for understanding effect systems as a type extension for managing free variables. This perspective provided insight into a stepwise approach for the design and construction of an effect system as a combination of two properly-designed language features: implicit parameters and type-based escape analysis. We explored the design space of the implicit parameters feature and introduced a novel type-based escape analysis technique that relies exclusively on subtyping and bounded polymorphism. Finally, we discussed several approaches that could enhance the practicality of effect systems for mainstream programming languages.

The prototype implementation of the presented effect system is available online.[5]

[5] https://github.com/penguin-boxx/opo-effect-system

References

[1] Marat Akhin and Mikhail Belyaev. 2021. Kotlin language specification. *Kotlin Language Specification* (2021).

[2] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29. https://doi.org/10.1145/3371116

[3] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing types. *ACM Transactions on Programming Languages and Systems* 45, 4 (2023), 1–52. https://doi.org/10.1145/3618003

[4] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527320

[5] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428194

[6] R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of functional programming* 17, 6 (2007), 687–730. https://doi.org/10.1017/S0956796807006259

[7] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 423–437. https://doi.org/10.1145/3385412.3386003

[8] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*. 15–27. https://doi.org/10.1145/2976022.2976033

[9] Daniel Hillerström and Sam Lindley. 2018. Shallow effect handlers. In *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16*. Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

[10] Mark P Jones. 1995. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer, 97–136. https://doi.org/10.1007/3-540-59451-5_4

[11] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices* 48, 12 (2013), 59–70. https://doi.org/10.1145/2578854.2503791

[12] John Launchbury and Simon L Peyton Jones. 1995. State in haskell. *Lisp and symbolic computation* 8, 4 (1995), 293–341. https://doi.org/10.1007/BF01018827

[13] Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061* (2014). https://doi.org/10.48550/arXiv.1406.2061

[14] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 486–499. https://doi.org/10.1145/3009837.3009872

[15] Jeffrey R Lewis, John Launchbury, Erik Meijer, and Mark B Shields. 2000. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 108–118. https://doi.org/10.1145/325694.325708

[16] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 333–343. https://doi.org/10.1145/199448.199528

[17] Sam Lindley. 2018. Encapsulating effects. In *Dagstuhl Reports, Volume 8, Issue 4*. https://doi.org/10.4230/DagRep.8.4.104

[18] Anton Lorenzen, Leo White, Stephen Dolan, Richard A Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with modal memory management. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 485–514. https://doi.org/10.1145/3674642

[19] John M Lucassen and David K Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 47–57. https://doi.org/10.1145/73560.73564

[20] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. 103–104.

[21] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. 2021. Safer exceptions for Scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. 1–11. https://doi.org/10.1145/3486610.3486893

[22] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. *ACM Sigplan Notices* 45, 10 (2010), 341–360. https://doi.org/10.1145/1932682.1869489

[23] Young Gil Park and Benjamin Goldberg. 1992. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 116–127. https://doi.org/10.1145/143095.143125

[24] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. *ACM SIGPLAN Notices* 49, 9 (2014), 123–135. https://doi.org/10.1145/2692915.2628160

[25] Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied categorical structures* 11 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

[26] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical methods in computer science* 9 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[27] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. 98–113. https://doi.org/10.1145/3331545.3342595

[28] Alejandro Serrano, Marat Akhin, Nikita Bobko, Ilya Gorbunov, Mikhail Zarechenskii, and Denis Zharkov. 2024. Context parameters (KEEP-367). https://github.com/Kotlin/KEEP/blob/context-parameters/proposals/context-parameters.md. Design proposal for context parameters in Kotlin. Last updated: Nov 21, 2024. Accessed: 2025-06-23.

[29] Andrey Stoyan. 2024. Modern Effect Systems Overview. In *2024 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 1–7. https://doi.org/10.1109/ISPRAS64596.2024.10899144

[30] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal effect types. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1130–1157. https://doi.org/10.1145/3720476

[31] Cas van der Rest, Jaro Reinders, and Casper Bach Poulsen. 2022. Handling Higher-Order Effects. *arXiv preprint arXiv:2203.03288* (2022).

[32] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. 1–12. https://doi.org/10.1145/2633357.2633358

[33] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29. https://doi.org/10.1145/3408981

[34] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-class names for effect handlers. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 30–59. https://doi.org/10.1145/3563289

[35] Yichen Xu, Oliver Bračevac, Cao Nguyen Pham, and Martin Odersky. 2025. What's in the Box? *Ergonomic and Expressive Capture Tracking over Generic Data Structures (Extended Version),(Aug. 2025). doi* 10 (2025), 17. https://doi.org/10.48550/arXiv.2509.07609

[36] Yichen Xu and Martin Odersky. 2023. Formalizing Box Inference for Capture Calculus. *arXiv preprint arXiv:2306.06496* (2023). https://doi.org/10.48550/arXiv.2306.06496

[37] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C Myers. 2016. Accepting blame for safe tunneled exceptions. *ACM SIGPLAN Notices* 51, 6 (2016), 281–295. https://doi.org/10.1145/2980983.2908086

[38] Yizhou Zhang, Guido Salvaneschi, and Andrew C Myers. 2020. Handling bidirectional control flow. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428207

[39] Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual modal types for algebraic effects and handlers. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473580