Answer-Type Modification for Lexical Effect Handlers (Extended Abstract)

Kai Pischke* Yaozhu Sun* Xu Xue Taro Sekiyama

Lexical effect handlers differ from traditional dynamically scoped handlers in a crucial respect: each effect invocation explicitly identifies the handler that interprets it, whereas under dynamic scoping the runtime selects the nearest matching handler. This explicit identity facilitates the isolation of independent resources, but complicates static reasoning about control flow. A capability may target a handler several lexical regions away, and an invocation through that capability captures a trail of delimited continuations whose shape depends on that distance. We present an answer-type modification (ATM) type system that makes both the target and the captured trail explicit. Operation types describe the pre- and post-states of the captured continuation; subregion evidence identifies the target handler; and a distance-indexed answer-type context identifies the continuation layer that the operation may modify. We prove the type system sound and extend it with parametric polymorphism and refinement types. We also present an algorithmic system that reconstructs evidence and answer-type contexts, implement it on top of Lexa, and evaluate the implementation on two benchmark suites.

1 Lexical Effect Handlers

Algebraic effect handlers provide a modular interface for control effects such as state, I/O, exceptions, backtracking, coroutines, and threads [4]. An invocation suspends the current computation as a delimited continuation and gives it to a handler clause, which may resume it zero, one, or several times. This first-class control is useful, but makes state transitions harder to verify than ordinary calls.

The distinction between dynamic and lexical handlers is central to our work. With a traditional dynamically scoped handler, an invocation does not identify a particular handler. The runtime searches outward from the invocation and selects the nearest enclosing handler for the operation. Lexical handlers instead associate a first-class name, effect instance, or capability with a handler; the invocation carries that name and is interpreted by the named handler [5]. Thus lexical scope makes handler identity explicit and supports resource isolation, but a call can now cross intervening handlers that are unrelated to it.

The following fragment makes the distinction concrete. Two handlers implement the same file signature, but lexical instances `‘a` and `‘b` distinguish their resources:

```
handle ‘a in
  handle ‘b in
    let x = read ‘a () in
    let y = read ‘b () in
    let _ = close ‘a () in
    let _ = close ‘b () in
    x + y
  with file_handler "2.txt"
with file_handler "1.txt"
```

The reads go to different providers: `read ‘a` reads from `"1.txt"` and `read ‘b` from `"2.txt"`; each close changes only its own descriptor from open to closed. Erasing the instances yields the traditional dynamic form: the nearest handler for `"2.txt"` receives both reads, and the program closes `"2.txt"` twice. A set such as {`read`,`close`} records neither association.

Thus lexical verification has two coupled obliga-

* Kai Pischke and Yaozhu Sun contributed equally to this work. Kai Pischke (kai.pischke@cs.ox.ac.uk), Oxford University; Yaozhu Sun (sun@nii.ac.jp) and Taro Sekiyama (tsekiyama@acm.org), National Institute of Informatics; Xu Xue (xxue@cs.hku.hk), The University of Hong Kong.

tions: *effect safety*, which proves that a capability reaches the handler it names, and a continuation protocol, which permits `read` only in the open state and makes `close` produce the closed state. Our ATM system connects these obligations.

2 How Answer-Type Modification Fits the Lexical Setting

2.1 Answer Types as Continuation Protocols

Answer-type modification (ATM) types not only a computation's result but also the answer expected by its surrounding continuation [1]. We write a computation type as

$$T \,/\, C_{\text{in}} \Rightarrow C_{\text{out}},$$

where T is the result, and C_{in} and C_{out} are the answer types before and after it. The notation is state-like: a computation can require one continuation state and guarantee another.

For the file example, suppressing the rest of the continuation, the two operations have the following protocol:

$$\begin{aligned} \mathsf{read} &: \mathsf{Unit} \to \mathsf{Int}/\mathsf{Open} \Rightarrow \mathsf{Open}, \\ \mathsf{close} &: \mathsf{Unit} \to \mathsf{Unit}/\mathsf{Open} \Rightarrow \mathsf{Closed}. \end{aligned}$$

A `read` clause must resume its continuation with a value of state type Open, while a `close` clause uses state type Closed; the operation types therefore check providers as well as clients.

Sequential composition connects the protocol: after `close` the intermediate answer type is Closed, whereas a following `read` requires Open, so the order is rejected even through a function call or handler clause. ATM alone does not say which continuation layer a lexical call modifies; regions, evidence, distance, and answer-type contexts supply that missing structure.

2.2 Lexical Names Determine the Target Handler

Every lexical handler introduces a region. A capability has type $\mathsf{Cap}\ \rho\ o$, where ρ is its provider's region and o its operation type. An invocation is written schematically as

$$\mathsf{do}\ x\,[e; \mathbb{C}](v),$$

where e is subregion evidence and $\mathbb{C}$ an answer-type context. The evidence e proves $\rho \sqsubseteq \rho_0$: the current region ρ is nested inside the provider region ρ_0 named by the capability. Thus e is a static witness that this dispatch is permitted from the current lexical nesting; it is not a dynamic search for a nearest handler.

This is not optional annotation: an operation type alone does not show that the call reaches the handler whose state it describes. For `read ‘b` the evidence is reflexive; for `read ‘a` it crosses the region for `‘b` and reaches the outer region for `‘a`. The same signature therefore has different targeting evidence at the two sites.

Dynamic handlers hide this issue: the target is always nearest, so the evidence is trivial. Lexical capabilities can skip an unrelated nearest handler; effect safety is therefore a relation between call-site and provider regions, not a property of the operation name.

2.3 Distance Determines Which Continuations Are Modified

Skipping a handler also changes the captured continuation. With an inner handler H_1 and outer handler H_2, a call handled by H_1 captures the nearest layer; a call naming H_2 crosses H_1's delimiter. The continuation passed to H_2 is then a fused trail fragment, with the rest as a contextual metacontinuation. The operation type must describe both.

We record crossed handler boundaries as a distance d. An answer-type context is a type-level expression with a hole: $\mathbb{C}[C]$ places C at the layer modified by the operation. Its well-formedness kind says that, for distance d, it describes exactly the fused d-boundary trail. In the example, the inner call has $d = 0$ and the outer call $d = 1$.

At a high level, the invocation rule combines these ingredients. If the operation has type

$$T \to T' \,/\, C_1 \Rightarrow C_2,$$

and evidence e proves that the current region reaches the capability's region, then $d = \mathsf{dist}(e)$ selects the answer-type context. The caller's initial and final answer types must be splittable using the same context:

$$\mathbb{C}[C_1] \quad \Rightarrow \quad \mathbb{C}[C_2].$$

The operation clause receives the reified continuation described by $\mathbb{C}$, together with the contextual metacontinuation outside the hole. The rule thus checks reachability, the state transition, and the exact layer that may change. Dynamic handlers need no distance index because their target is fixed to the nearest delimiter.

3 The Declarative Calculus

3.1 Handlers, Capabilities, and Regions

The core language contains values, computations, capabilities, and a lexical `handle-with` construct, represented at the calculus level as

$$\mathsf{handle}\ \{[\alpha;\beta](\overline{x}) \Rightarrow M\}\ \mathsf{with}\ H.$$

Here α is the fresh region introduced for the handler, and β is subregion evidence connecting that handler region α to the surrounding body region. Capabilities $\overline{x}$ carry α and the operation types inferred from H.

Operationally, the capability selects the handler, the semantics captures the continuation up to it, and the matching clause receives the argument and continuation. Unlike dynamic search, this makes the clause's continuation protocol part of its type.

3.2 The Two Key Typing Rules

The declarative system makes both obligations explicit. Let ρ be the current region and ρ_0 the capability's provider region. The two rules are shown together in Fig. 1.

$$\frac{\Gamma \vdash x : \mathsf{Cap}\ \rho_0\ (T \to T'/C_1 \Rightarrow C_2) \qquad \Gamma \vdash e : \mathsf{Ev}[\rho \overset{d}{\sqsubseteq} \rho_0] \quad \Gamma \vdash \mathbb{C} : \mathsf{ATC}[d] \quad \Gamma \vdash v : T}{\Gamma \mid \rho \vdash \mathsf{do}\ x[e;\mathbb{C}](v) : T'/\mathbb{C}[C_1] \Rightarrow \mathbb{C}[C_2]}\ \textsc{Do}$$

$$\frac{\dfrac{\Gamma, \alpha : \mathsf{Reg}, \quad \beta : \mathsf{Ev}[\alpha \overset{1}{\sqsubseteq} \rho]}{x : \mathsf{Cap}\ \alpha\ o \mid \alpha \vdash M : T/C_1 \Rightarrow C_2} \qquad \Gamma \mid \rho \vdash H : (T \to C_1) \uplus \overline{o}}{\Gamma \mid \rho \vdash \mathsf{handle}\{[\alpha;\beta](\overline{x}) \Rightarrow M\}\ \mathsf{with}\ H : C_2}\ \textsc{Handle}$$

Fig. 1 Key declarative typing rules.

In *Do*, e proves reachability and $d = \mathsf{dist}(e)$ selects the layer; the same $\mathbb{C}$ surrounds C_1 and C_2, so only that state changes. *Handle* introduces α, its one-step evidence β, and capabilities for M, establishing handler identity and protocol together.

In the file example, `"1.txt"` and `"2.txt"` contribute separate answer-type components. The inner capability uses $d = 0$ and changes only `"2.txt"`; the outer capability crosses the inner region and changes `"1.txt"` while preserving `"2.txt"`. This is why the lexical program closes the two descriptors independently. After erasing instances, dynamic dispatch sends both calls to `"2.txt"`, making the second close invalid.

To make the separation explicit, the relevant part of the derivation is:

call	e	d	modified component
read b	Refl	0	`2.txt` : Open ⇒ Open
close a	Step(Refl)	1	`1.txt` : Open ⇒ Closed

For the first row, the answer-type context has a hole at the nearest `"2.txt"` component. For the second, the context has a hole at `"1.txt"` and retains the inner `"2.txt"` component in its surrounding trail. Thus the *Do* premises prove not only that the operation is in scope, but also that its pre/post-state is attached to the intended resource. If the instances are erased, both rows would have nearest-handler evidence and $d = 0$; the second close would then be a second close of `"2.txt"`, not a close of `"1.txt"`. This small derivation is the reason lexical handler identity cannot be recovered from an ordinary effect set.

3.3 Soundness

We prove soundness by progress and subject reduction. Progress relies on evidence to guarantee that a well-typed invocation names an in-scope handler; subject reduction preserves the ATM type through reduction, dispatch, capture, and resumption. In the lexical cases, evidence guarantees the selected handler and the answer-type context guarantees the continuation expected by its clause—obligations left implicit by effect-set typing.

4 An Algorithmic System and an Implementation

The algorithmic system hides evidence from programmers. It stores a tree of lexical regions and each capability's provider region. At a call, it follows the enclosing links from the current region to that provider: this is compile-time path bookkeeping, not runtime handler search.

A call at the provider has path length zero; a nested call composes each enclosing edge into evidence, and the number of edges is d. Thus the inner and outer capabilities in our example have $d = 0$ and $d = 1$. If polymorphism hides the target, the compiler postpones the reachability check until instantiation rather than guessing.

For an invocation, the compiler checks the argument and the capability's protocol, uses the path to compute d, then selects the d-th answer-type

layer. It checks the pre/post-state there, preserves other layers, and elaborates the call with the reconstructed evidence and context. Missing paths and insufficient layers are rejected unless explicitly annotated.

We prove elaboration soundness: an accepted evidence-free program elaborates to a term well typed in the declarative calculus. Thus the convenient source language still discharges the distance-sensitive lexical obligations rather than replacing them with an untracked dynamic convention.

The implementation extends Lexa, a language and compiler for lexical effect handlers [3]. On top of Lexa's stack-switching runtime, we extend the type system with computation types, regions, distances, answer-type contexts, polymorphism, and refinement checking. Handler labels serve as invocation and region names; in the common case `do x` is elaborated with reconstructed evidence and context, while an explicit form documents ambiguous cases.

Two benchmark suites cover ATM examples adapted to lexical handlers (nondeterminism, state, queues, I/O-like protocols, coroutines, and transactions) and Lexa programs (recursion, iterators, early exits, logging, and cooperative threads). Valid programs are accepted and ill-typed variants rejected, showing that the distance/context machinery is implementable rather than merely declarative.

5 Extensions, Comparison, and Limits

Parametric polymorphism lets a function accept a caller's capability without fixing its handler. Plain ATM tracks continuation protocols, but cannot relate the value written to a cell to the value later read from it. Refinements make this value flow explicit: `set x` resumes with state x, while `get` returns the current state and passes it on unchanged. They thus verify value-level state invariants that plain ATM cannot express. This extension adapts refinement ATM to lexical handlers [2].

Dynamic handlers fix the nearest target and continuation layer. Lexical identity is a first-class choice: the target may be outer and the captured trail multi-layered. Evidence proves the target, distance counts fused layers, and the answer-type context types that trail; ATM supplies the state transition. Removing any one of these ingredients loses resource identity or nesting sensitivity.

The separation yields distinct failures: no evidence means an out-of-scope target; an ATM mismatch means a protocol error; a shallow answer type means an insufficient continuation trail; and a wrong state type at resumption means a provider error. Dynamic effect sets tend to collapse these cases into an unstructured operation occurrence, whereas our rules distinguish these cases.

The approach complements ownership-based typestate: ATM expresses protocols and handler continuations beyond linear-use conditions, but does not enforce affine capabilities or one-shot continuations; correspondence between the static state type and actual resource state remains a programmer invariant. The system verifies handler selection and typestate transitions, not termination or liveness. Modular contexts and a typed-CPS correspondence remain open.

6 Conclusion

Lexical effect handlers make identity explicit, enabling resource isolation but exposing a problem absent from nearest-handler dispatch. Our ATM system tracks resource pre/post-states and the trail selected by each capability through evidence, distances, and answer-type contexts. The declarative system is sound, the algorithm reconstructs this structure, and the Lexa implementation demonstrates the design on realistic programs.

References

[1] Danvy, O. and Filinski, A.: A Functional Abstraction of Typed Contexts, Technical report, DIKU, University of Copenhagen, 1989.

[2] Kawamata, F., Unno, H., Sekiyama, T., and Terauchi, T.: Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers, *Proceedings of the ACM on Programming Languages*, Vol. 8, No. POPL(2024), pp. 115–147.

[3] Ma, C., Ge, Z., Lee, E., and Zhang, Y.: Lexical Effect Handlers, Directly, *Proceedings of the ACM on Programming Languages*, Vol. 8, No. OOPSLA2(2024), pp. 1670–1698.

[4] Plotkin, G. D. and Pretnar, M.: Handlers of Algebraic Effects, *Programming Languages and Systems*, 2009, pp. 80–94.

[5] Xie, N., Cong, Y., Ikemori, K., and Leijen, D.: First-class names for effect handlers, *Proceedings of the ACM on Programming Languages*, Vol. 6, No. OOPSLA2(2022), pp. 30–59.