Abstraction-Safe Effect Handlers via Tunneling: Technical Report

YIZHOU ZHANG, Cornell University, USA
ANDREW C. MYERS, Cornell University, USA

ABSTRACT

Algebraic effect handlers offer a unified approach to expressing control-flow transfer idioms such as exceptions, iteration, and async/await. Unfortunately, previous attempts to make these handlers type-safe have failed to support the fundamental principle of modular reasoning for higher-order abstractions. We demonstrate that abstraction-safe algebraic effect handlers are possible by giving them a new semantics. The key insight is that code should only handle effects it is aware of. In our approach, the type system guarantees all effects are handled, but it is impossible for higher-order, effect-polymorphic code to accidentally handle effects raised by functions passed in; such effects tunnel through the higher-order, calling procedures polymorphic to them. By contrast, the possibility of accidental handling threatens previous designs for algebraic effect handlers. We prove that our design is not only type-safe, but also abstraction-safe. Using a logical-relations model that we prove sound with respect to contextual equivalence, we derive previously unattainable program equivalence results. Our mechanism offers a viable approach for future language designs aiming for effect handlers with strong abstraction guarantees.

1 INTRODUCTION

Algebraic effects [5, 38, 39] have developed into a powerful unifying language feature, shown to encompass a wide variety of other important features that include exceptions, dynamically scoped variables, coroutines, and asynchronous computation. Although some type systems make algebraic effects *type-safe* [4, 28, 30], we argue in this paper that algebraic effects are not yet *abstraction-safe*: details about the use of effects leak through abstraction boundaries.

As an example, consider the higher-order abstraction `map`, which applies the same function to each element in a list:

```
map[X,Y,E](l : List[X], f : X → Y throws E) : List[Y] throws E
```

In general, the computation embodied in the functional argument `f` may be effectful, as indicated by the clause `throws E` in the type of `f`. To make it reusable, `map` is defined to be polymorphic over the latent effects `E` of `f`, and propagates any such effect to its own caller.

The `map` abstraction can be implemented in many different ways; modularity is preserved if clients cannot tell which implementation is hiding behind the abstraction boundary. It would thus be surprising if two implementations of this `map` abstraction behaved differently when used in the same context. However, current semantics of algebraic effects allow a client to observe different behaviors—and to distinguish between the two implementations—when one of the implementations happens to use algebraic effects internally.

For example, suppose an implementation of `map` traverses the list using an iterator object. The iterator throws a `NoSuchElement` exception when it reaches the end of the list, and the implementation handles it accordingly. If the client function `f` also happens to throw `NoSuchElement`, the implementation may handle—*by accident*—an effect it is not designed to handle. By breaking the implementation of `map` in this way, such a client thereby improperly observes internals of its implementation. This violation of abstraction is also a failure of modularity.

We contend that this failure is a direct consequence of the dynamic semantics of algebraic effect handlers. Intuitively, for Reynolds' Abstraction Theorem [41] (also known as the Parametricity Theorem [45]) to hold for a language with type abstraction (such as System F), polymorphic functions cannot make decisions based on the types instantiating the type parameters. Analogously, parametricity of effect polymorphism demands that an effect-polymorphic function should not make decisions based on the effect it is instantiated with. Yet the dynamic nature of algebraic effects runs afoul of this requirement: an effect is handled by searching the dynamic scope for a handler that can handle the effect. To restore parametricity, we propose to give algebraic effects a new semantics based on *tunneling*:

> *Algebraic effects can be handled only by handlers that are statically aware of them; otherwise, effects* tunnel *through handlers.*

This semantics provides sound modular reasoning about effect handling, while preserving the expressive power of algebraic effects.

For a formal account of abstraction safety, the typical syntactic approach to type soundness no longer suffices, because it is difficult to syntactically track type-system properties that are deeper than subject reduction [6, 14, 31, 46]. By contrast, a semantic approach that gives a relational interpretation of types can be applied to the harder problem of reasoning about program refinement and equivalence. Therefore, a prime result of the present paper is a semantic type-soundness proof for a core language with tunneled algebraic effects. To this end, we define a step-indexed, biorthogonal logical-relations model for the core language, giving a relational interpretation not just to types, but also to effects. We show this logical-relations model offers a sound and complete reasoning process for proving contextual refinement and equivalence. Effectful program fragments can then be rigorously proved equivalent, supporting reasoning about the soundness of program transformations. We proceed as follows:

- We illustrate the problem of accidentally handled effects in Section 2, clarifying the observation that algebraic effect handlers violate abstraction.
- We present tunneled algebraic effects in Section 3. Tunneling causes no significant changes to the usual syntax of algebraic effects; it changes the dynamic semantics of effects but does not lose any essential expressive power.
- We define the operational and static semantics of tunneling via a core language (Section 4).
- In Section 5, we give a logical-relations model for the core language. We establish important properties of the logical relation, including parametricity and soundness with respect to contextual refinement. These results, checked using Coq, make rigorous the claim that tunneled algebraic effects are abstraction-safe.
- We demonstrate the power of the logical relation in Section 6 by proving program equivalence. As promised, effect-polymorphic abstractions in the core language hide their use of effects.
- We survey related work (Section 7) and conclude (Section 8).

2 ALGEBRAIC EFFECTS AND ACCIDENTAL HANDLING

Algebraic effects are gaining popularity among language designers because they enable statically checked, programmer-defined control-flow transfer. Legacy language abstractions for control flow, including exceptions, yielding iterators, and async/await, become just instances of algebraic effects.

We illustrate the problems with algebraic effects in the setting of a typical object-oriented language, like Java, C#, and Scala, that has been extended with algebraic effects and effect polymorphism. Despite this object-oriented setting, the problems we identify and the solution we propose are broadly applicable to languages with algebraic effects or with mechanisms subsumed by algebraic effects.

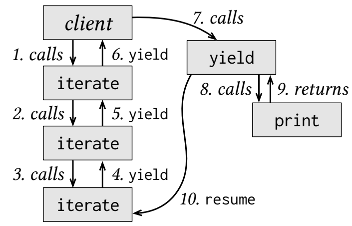
2.1 Algebraic Effects and Handlers

The generality of algebraic effects comes from the ability to define an *effect signature* whose implementations are provided by *effect handlers*. An effect signature defines one or more *effect operations*. For example, the code below

```
effect Yield[X] {
  yield(X) : void
}
```

```
val tr : Tree[int] = ...
try { iterate(tr) }
with yield(x) {
    print(x)
    resume()
}
```

(a)



(b)

Figure 1. (a) Client code iterating over a binary tree. (b) A stack diagram showing the control flow.

defines an effect signature named `Yield`, parameterized by a type variable `X`. This signature contains only one operation, `yield`, and invoking this operation requires a value of type `X`. This `Yield` effect can be used for declarative definitions of iterators. For example, the function `iterate` is an in-order iterator for binary trees:

```
interface Tree[X] {
  value() : X
  left() : Tree[X]
  right() : Tree[X]
}
```

```
iterate[X](tr : Tree[X]) : void throws Yield[X] {
  iterate(tr.left())
  yield(tr.value())
  iterate(tr.right())
}
```

Invoking an effect operation has the corresponding effect. In the example, the `iterate` function invokes the `yield` operation, so it has the effect `Yield[X]`. Static checking of effects requires that this effect be part of the function's type, in its `throws` clause.

Traversing a tree using the effectful `iterate` function uses the help of an effect handler (Figure 1a). The effectful computation is surrounded by `try {...}`, while the handler follows `with` and provides an implementation for each effect operation. In this example, the implementation of `yield` first prints the yielded integer, and resumes the computation in the `try` block.

The implementation of an effect operation has access to the *continuation* of the computation in the corresponding `try` block. This continuation, denoted by the identifier `resume`, takes as an argument the result of the effect operation, and when invoked, resumes the computation at the invocation of the effect operation in the `try` block. Because the result type of `yield` is `void`, the call to `resume` accepts no argument. Figure 1b visualizes the control flow under this resumptive semantics using a stack diagram.

The handling code of Figure 1a is actually syntactic sugar for code declaring an anonymous handler:

```
try { iterate(tr) }
with new Yield[int]() {
  yield(x : int) : void { print(x); resume() }
}
```

The sugared form in Figure 1a requires the name `yield` to be unambiguous in the context. It is also possible to define standalone handlers instead of inlining them. Handlers can also have state. For example, handler `printInt`, defined separately from its using code, stops the iteration after 8 rounds:

```
handler printInt for Yield[int] {
  var cnt = 0 // State of the handler
  yield(x : int) : void {
    if (cnt < 8) { print(x); ++cnt; resume() }
  }
}
```

```
// Using code allocates a handler object
// with state cnt initialized to 0
try { iterate(tr) }
with new printInt()
```

```
fsize1[X,E](tr : Tree[X], f : X → bool / E) :
int / E {
    val num = 0
    try { fiterate(tr, f) }
    with yield(x : X) : void {
      ++num; resume()
    }
    return num
}
```

(a)

```
fsize2[X,E](tr : Tree[X], f : X → bool / E) :
int / E {
    val lsize = fsize2(tr.left(), f)
    val rsize = fsize2(tr.right(), f)
    val cur = f(tr.value()) ? 1 : 0
    return lsize + rsize + cur
}
```

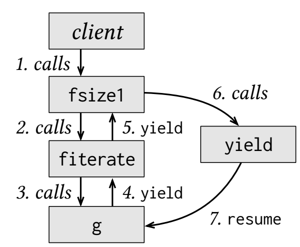
(b)

Figure 2. Two implementations of a higher-order abstraction. The intended behaviors of these two implementations are the same: returning the number of elements satisfying a predicate in a binary tree.

```
val fsize = ... // The right-hand side is either fsize1 or fsize2
val g = fun(x : int) : bool / Yield[int] { yield(x) ; f(x) }

try { fsize(tr, g) }
with yield(x : int) : void {
    ... // do something with x
    resume()
}
```

(a)



(b)

Figure 3. (a) A client that can distinguish between `fsize1` and `fsize2`, two supposedly equivalent implementations of the same abstraction. (b) Snapshot of the stack when `fsize1` accidentally handles an `Yield` effect raised by applying g.

Effect Polymorphism. Higher-order functions like `map` accept functional arguments that are in general effectful. Such higher-order functions are therefore polymorphic in the effects of their functional argument. Language designs for effects typically include this kind of polymorphism to allow the definition of reusable generic abstractions [23, 28, 30, 42]. As an example, consider a filtering iterator that yields only those elements satisfying a predicate `f` that has its own effects `E`.

```
fiterate[X,E](tr : Tree[X], f : X → bool / E) : void / Yield[X], E {
  foreach (x : X) in tr
    if (f(x)) { yield(x) }
}
```

Here we introduce "/" as a shorthand for `throws`. The higher-order function is parameterized by an *effect variable* `E`, which is the latent effect of the predicate `f`. The implementation iterates over the tree and yields elements that test true with `f`. Because it invokes `yield` and `f`, its effects consist of both `Yield[X]` and `E`.

2.2 Accidentally Handled Effects Violate Abstraction

Suppose we want a higher-order abstraction that computes the number of tree elements satisfying some predicate. It can be implemented by counting the elements yielded by `fiterate`, as shown in Figure 2a. The same abstraction can also be implemented in a recursive manner, as shown in Figure 2b. We would hope that these implementations are *contextually equivalent*, meaning that they can be interchanged freely without any client noticing a difference.

Unfortunately, there do exist clients that can distinguish between the two implementations, as shown in Figure 3a. This client code interacts with the abstraction whose implementation is provided either by `fsize1` or by `fsize2`, and uses a function named `f` as the predicate. But it also does something else with each element that `f` is applied to, using the help of an effect handler: it wraps `f` in another function `g` (line 2), which, before calling `f`, yields the element to a handler that does the extra work (line 5). The client passes to the abstraction the wrapper `g`, which is eventually applied somewhere down the call chain. This application of `g` raises an `Yield[int]` effect, which the programmer would expect to be propagated back to the client code and handled at lines 4–7.

However, the programmer will be unpleasantly surprised if the client uses the implementation provided by `fsize1`. At the point where the effect arises, the runtime searches the dynamic scope for a handler that can handle the effect. Because the nearest dynamically enclosing handler for `Yield[int]` is the one in `fsize1` (lines 5–7 in Figure 2a), the effect is unexpectedly intercepted by this handler, incorrectly incrementing the count. Figure 3b shows the stack snapshot when this accidental handling happens.

By contrast, the call to `fsize2` behaves as expected. Hence, two well-typed, type-safe, intuitively equivalent implementations of the same abstraction exhibit different behaviors to the same client. Syntactic type soundness is preserved—neither program gets stuck during execution—but the type system is not doing its job of enforcing abstraction.

The above example demonstrates a violation of abstraction from the implementation perspective, but a similar story can also be told from the client perspective: two apparently equivalent clients can make different observations on the same implementation of an abstraction. For example, consider the following two clients of `fsize1`: one looks like Figure 3a but with line 5 left empty, and the other is simply `fsize1(tr, f)`.

The handling of the `Yield` effect in the first client ought to amount to a no-op, so the two programs would be equivalent. Yet the equivalence does not hold because of the accidental handling of effects in the first program. This client perspective shows directly that the usual semantics of algebraic effect handling fails to comply with Reynolds' notion of relational parametricity [41], which states that applications of a function to related inputs should produce related results.

Prior efforts based on effect rows and row polymorphism have aimed to prevent client code from meddling with the effect-handling internals of library functions [7, 27]. Notably, recent work by Biernacki et al. [7] has shown relational parametricity for a core calculus with algebraic effects, but the type system compromises on the expressiveness of effect subsumption and relies on extra programmer annotations. For example, under their typing rules, function `fsize1` would not type-check unless (a) its signature mentioned the `Yield` effect, thereby exposing the implementation detail that `fsize1` handles `Yield` internally:

```
fsize1[X,E](tr : Tree[X], f: X → bool / { Yield[X], E } ) : int / E
```

or (b) a special "lift" operator is inserted at the place where `f` is applied in `fiterate`.

3 TUNNELED ALGEBRAIC EFFECTS

Just as algebraic effect handlers arose as a generalization of exception handlers [39], we build on the insight of Zhang et al. [47], who argue that *tunneled exceptions* make exceptions safer through a limited form of exception polymorphism. We show that tunneling can be generalized to algebraic effects broadly along with the general form of effect polymorphism presented in Section 2.1.

Tunneled algebraic effects address the problem of accidental handling. Despite this increase in safety, there is no increase in programmer effort. In fact, with the new tunneling semantics in effect, the examples from Section 2.2 become free of accidental handling, with *no* syntactic changes required.

Consider the version of Figure 3a that resulted in accidental handling of effects (i.e., the version that uses `fsize1`). Under the new semantics, the `Yield` effect raised by applying `g` is tunneled straightaway to the client code, without being intercepted by the intermediary contexts. Figure 4 shows the stack snapshot when this tunneling happens.

3.1 Tunneling Restores Modularity

This tunneling semantics enforces the modular reasoning principle that handlers should only handle effects they are locally aware of. In the example, the intermediary contexts, `fsize1` and `fiterate`, are polymorphic in an effect variable that represents the latent effects of their functional arguments. So they ought to be *oblivious* to whatever effect applying `g` might raise at run time. The modular reasoning principle hence prohibits handlers in these intermediary contexts from capturing any dynamic instantiations of the effect variable; accidental handling is impossible.

The client code, by contrast, is locally *aware* that applying `fsize1` to `g` manifests the latent effect of `g`. The modular reasoning principle thus requires that the client code provide a handler for this effect in order to maintain type safety.

The lack of modularity in the presence of higher-order functions is an inherent problem of language mechanisms based on some form of dynamic scoping, many of which are subsumed by algebraic effects. Among such effects, the one that most famously conflicts with modular reasoning is perhaps dynamically scoped variables.

Dynamically scoped variables increase code extensibility, as exemplified by the TEX programming language [26], because they act as implicit parameters that can be accessed—and overridden—in their dynamic extents. But their unpredictable semantics prevents wider adoption. In particular, a higher-order function may accidentally override variables that its functional argument expects from the dynamic scope, a phenomenon known in the Lisp community as the "downward funarg problem" [43]. This problem with dynamically scoped variables is an instance of accidental handling.

Fortunately, tunneling offers a solution broadly applicable to all algebraic effects, including dynamically scoped variables and exceptions. We illustrate this solution through an example involving the tunneling of multiple effects.

3.2 Tunneling Preserves the Expressivity of Dynamic Scoping Safely

Consider the Visitor design pattern [20], which recursively traverses an abstract syntax tree (AST). Visitors often keep intermediate state in some associated *context*. For example, a type-checking visitor would use a typing environment as the context, while a pretty-printing visitor would use a context to keep track of the current indentation level. The state in such contexts is essentially an instance of dynamic scoping. Moreover, the type-checking visitor may expect the context to handle typing errors, while the pretty-printing visitor needs the context to handle I/O exceptions. A common `Visitor` interface is therefore unable to capture this variability in the notion of context. So either uses of the Visitor pattern are limited to settings that do not need context, or the programmer has to resort to error-prone workarounds.

One such workaround is to capture context information as mutable state. However, recursive calls to the visitor often need to update context information. So side effects need to be carefully undone as each recursive call returns; otherwise, subtrees yet to be visited would not have the right context information.

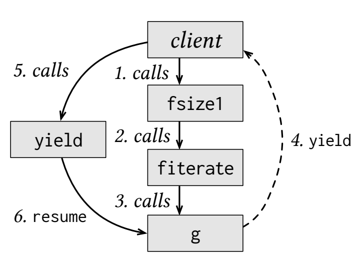


Figure 4. Snapshot of the stack when a `Yield` effect raised by applying `g` is tunneled to the client code.

```
1  interface Visitor[E]  {
2    visit(While) : void/E
3    visit(Assign) : void/E
4    ...
5  }

6  interface While extends Stmt {
7    cond() : Expr
8    body() : Stmt
9    accept[E](v : Visitor[E]) : void/E
10     { v.visit(this) }
11   ...
12 }

13 effect Val[X]
14   { get() : X }      // Immutable variables
15 effect Var[X] extends Val[X]
16   { put(X) : void }  // Mutable variables

17 effect IOExc { throw() : void }

18 print(s : String) : void / IOExc {...}
19 indent(l : int) : void / IOExc {...}

20 class pretty for Visitor[{Val[int],IOExc}]{
21   visit(w : While) : void / _ {   // Infers effects
22     val l = get()  // Current level of indentation
23     indent(l)                 // Print indentation
24     print("while ")
25     w.cond().accept(this)
26     print("\n")
27     try { w.body().accept(this) }
28     with get() : int {
29       resume(l + 1)  // Increment indentation level
30     }
31   }
32   ...
33 }

34 try {
35   val v = new pretty()
36   program.accept(v)
37 } with {
38   get() : int { resume(0) }
39   throw() : void {...}
40 }
```

Figure 5. Using tunneled algebraic effects to provide access to the context for visitors.

Tunneled algebraic effects provide the expressive power needed to address this quandary, without incurring the problems of dynamic scoping. Figure 5 shows a pretty-printing visitor defined using tunneled algebraic effects. The `Visitor` interface (lines 1–5) is generic with respect to the effects of the visitor methods. AST visitors can all implement this interface but provide their own notions of context. For the pretty-printer, indentation is modeled as an (immutable) dynamically scoped variable, whose effect signature is given on lines 13–14. This signature can be extended to support mutability (lines 15–16), though it is not needed by this example. The visitor also uses methods `print` and `indent` (lines 18 and 19), which can raise I/O exceptions.

Pretty-printing `While` loops (lines 21–31) manipulates the dynamic scope. To properly indent, the current indentation level is obtained from the dynamically scoped variable by invoking the effect operation `get` (line 22). The loop body is printed using the same visitor, but with an updated indentation level. This overriding of the dynamically scoped variable is done by providing a new handler for the recursive visit of the loop body (lines 27–30). The initial level of indentation is provided by the client code on line 38.

Figure 6 visualizes the propagation of a `Yield[int]` effect and an `IOExc` exception raised when visiting a loop body. Notice that these effects tunnel through the effect-polymorphic `accept` methods. So even if any of the `accept` methods handled effects internally, they would not be able to intercept the effects passing by.

3.3 Accomplishing Tunneling by Statically Choosing Handlers

The modular reasoning principle requires that it be possible to reason statically about which handler is used for each invocation of effect operations. Accordingly, the language mechanism for accomplishing tunneling requires that an effect handler be given whenever an effect operation is invoked. As we show below, such a handler can take the form of a concrete definition or of a *handler variable*, and does not have to be provided explicitly in typical usage.

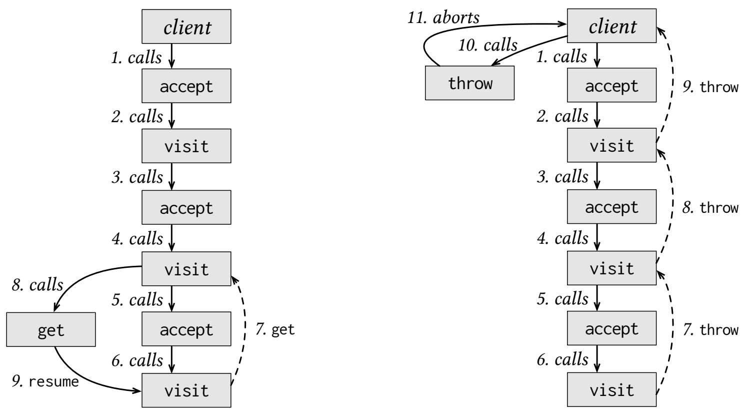


Figure 6. Left: stack snapshot at the point when printing the loop body asks for the current indentation level. Right: stack snapshot when an I/O exception is raised while printing the loop body.

The effect-handling code on the left is actually shorthand for the code on the right, which explicitly names the exception handler to use:

```
try { throw() }
with throw() { ... }
```

```
try { H.throw() }
with H = new IOExc() {
  throw() : void { ... }
}
```

The handler with a concrete definition is given the name `H`, and the invocation `H.throw()` indicates that `H` is chosen explicitly as the handler for the effect operation.

While the `try-with` construct introduces bindings of handlers with concrete definitions, mentions of effect names in method, interface, or class headers introduce bindings of *handler variables*. For example, the `iterate` method from Section 2.1 mentions `Yield[X]` in its `throws` clause:

```
iterate[X](tr : Tree[X]) : void / Yield[X] { ... }
```

So `iterate` is desugared using explicit parameterization with a handler variable named `h`:

```
iterate[X, h : Yield[X]](tr : Tree[X]) : void / h {
  iterate[X, h](tr.left())
  h.yield(tr.value())
  iterate[X, h](tr.right())
} // Uses of the handler variable are highlighted
```

The method is polymorphic over a handler for `Yield[X]`, and the effectful computation in its body is handled by this handler.

Inferring omitted handlers. Naming the handler might seem verbose, but does not create a burden on the programmer: when programs are written using the usual syntax, the choice of handler is obvious, so the language can always figure out what is omitted.

To map a program written in the usual syntax into one in which the choice of handler is explicit, two phases of rewriting are performed: desugaring, and resolving omitted handlers. Desugaring involves

(a) introducing explicit bindings for concrete handler definitions and explicit handler-variable bindings for handler polymorphism, and
(b) identifying where handlers are omitted and must be resolved—namely at invocation sites of effect operations and of handler-polymorphic abstractions.

Once the program is desugared, an omitted handler for some effect signature (or effect operation) is *always* resolved to the nearest lexically enclosing handler binding for that signature (or operation).

In the examples above, the concrete handler definition `H` is the closest lexically enclosing one for `IOExc`, and the handler variable `h` is the closest lexically enclosing one for `Yield[X]`. So when they are omitted in the program text, the language automatically chooses them as handlers for the respective effects.

Tunneling. Tunneling falls out naturally. Performing the rewriting discussed above on the example in Figure 3a yields the following program:

```
val fsize = ...
val g = fun[h : Yield[int]](x : int) : bool / h { h.yield(x); f(x) }
try { fsize(tr, g[H]) }
with H = new Yield[int]() { yield(x : int) : void { ... } }
```

When g is passed to the higher-order function, its handler variable is substituted with the locally declared handler H, the closest lexically enclosing one for Yield[int]. As a result, the invocation of the effect operation in g will unequivocally be handled by H, rather than being intercepted by some handler declared in an intermediary context.

As another example, class pretty in Figure 5 is actually parameterized by two handler variables ind and io representing the dynamically scoped indentation level and the handling of I/O exceptions:

```
class pretty[ind : Val[int], io : IOExc] for Visitor[{ind,io}] {
  visit(w : While) : void / {ind,io} {
    ...
    try { w.body().accept[{H,io}](this[H, io]) }
    with H = new Val[int]() {
      get() : int { resume(l+1) }
    }
    ...
  }
  ...
}
```

For the code that visits the loop body (i.e., line 27 of Figure 5, whose full form is also shown above), two handlers for Val[int] are lexically in scope—the handler variable ind and the handler definition named H. The closest lexically enclosing one is chosen, so loop bodies are visited using an incremented indentation level. Notice that the this keyword is actually a handler-polymorphic value, so it is possible to recursively invoke the visitor while overriding the handler. For the handling of I/O exceptions, the handler variable io is the only applicable handler lexically in scope. Both kinds of effects are guaranteed not to be captured by the effect-polymorphic accept methods.

Disambiguating the choice of handler. Although explicitly naming handlers is not necessary in most cases, the ability to specify handlers explicitly adds expressivity. For example, in their recent work on using algebraic effects to encode complex event processing, Braćevac et al. (2018) describe a situation where different invocations of the same effect operation need to be handled by different surrounding handlers. The ability to explicitly specify handlers addresses this need.

3.4 Region Capabilities as Computational Effects

With the rewriting described in Section 3.3, it may seem superfluous to still statically track the effects of methods like iterate and g via throws clauses. After all, the desugared method signatures explicitly require a handler to be provided—it appears guaranteed that the effect of any call to iterate or g is properly handled.

However, programs would go wrong if these effects were ignored. Consider the program on the left of Figure 7, where the type system does not track the effect of g other than requiring a handler to be provided. In this example, g is passed to the (higher-order) identity function, and the result is stored into a local variable f. As with the fsize example, the handler to provide for g is resolved to the closest enclosing handler H. So when a Yield effect arises as a result of applying f to an integer (line 8), the

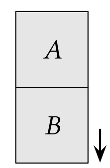
```
val f : int → void
val g = fun[h : Yield[int]](x : int) : void
        { ... h.yield(x) ... }
try { f = identity(g[H]) }
with H = new Yield[int]() {
  yield(x : int) : void { ... resume() }
}
f(0) // Invokes g[H](0) but causes a run-time error
```

```
val f : int → void
val g = fun[h : IOExc](x : int) : void
        { ... h.throw() ... }
try { f = identity(g[H]) }
with H = new IOExc()
         { throw() : void { ... } }
... // Unable to transfer control here when H finishes
return f // Run-time error if f is invoked later
```

Figure 7. Both programs go wrong as a result of the type system's not tracking the effect of g other than requiring a handler to be provided. Region capabilities (Section 3.4) address this issue.

handling code in `H` is executed. But `H` does not have a computation to resume: the current control state is no longer within a `try` block!

A similar problem happens when handlers do not resume—but rather abort—computations in `try` blocks, such as exception handlers. In the program on the right of Figure 7, `g` may throw an `IOExc` exception, and the computation in `g[H]` is returned to the caller. When an exception handler finishes, control ought to be transferred to the point immediately following the corresponding `try–with` statement. However, when `g[H]` is invoked later, raising an exception, the computation following `try–with` is no longer available when the exception handler `H` finishes execution, because the stack frame containing the computation has been deallocated.



We can view a `try–with` statement as marking a program point which, at run time, divides the stack into two regions. In the figure to the right, the stack grows downwards, and an effect is raised at the bottom of the stack. The two regions, A and B, represent the possible control-flow transfer targets when the handler finishes handling the effect: the upper region A is the computation to jump to if the handler aborts the computation in the `try` block, and the lower region B is the `try`-block computation possibly to be resumed.

To handle an effect thus requires the *capability* to access the stack regions. A `try–with` statement introduces a unique capability, which the corresponding handler holds within the `try` block. Capabilities must not be able to escape their corresponding `try` blocks; otherwise, they would refer to deallocated stack regions.

To this end, the type system tracks these stack-region capabilities as computational effects. In the example above, applying `g` needs the capability held by the handler variable `h`. So the effect of `g` is this capability, denoted by `h` in the `throws` clause of `g`:

```
val g = fun[h : Yield[int]](x : int) : void / h { ... h.yield(x) ... }
```

In the `try` block, the handler `H` provided by the enclosing `try–with` is used to substitute for the handler variable, so the expression `identity(g[H])`—and therefore `f`—must have type `int → void / H`, meaning that the capability held by `H` is needed to apply `f`. However, because `f` outlives the `try–with` that introduces this capability, the capability will be unavailable when `f` is applied. Fortunately, since capabilities are tracked statically, the type system rejects this program.

This capability-effect system is more expressive than previous approaches to effect polymorphism that use an escape analysis to prevent accesses to deallocated regions [36, 47]. In contrast to these approaches, we allow values with latent polymorphic effects to escape into (effect-polymorphic) data structures, as long as uses of the data structure do not outlive the corresponding stack regions. For example, `cachingFun` implements a function that caches the result of its application, and is polymorphic over the latent effects of that function:

```
// An effect-polymorphic data structure
class cachingFun[X,Y,E] for Fun[X,Y,E] {
  val f : X → Y/E
  cachingFun(f : X → Y/E) { this.f = f }
  apply(x : X) : Y/E { ... f(x) ... }
  ...
}
```

```
// Using code
val g = fun(x : int) : void / Yield { ... }
try {
  val f = new cachingFun(g)
  ... // Apply f
}
with yield(x : int) : void { ... resume() }
```

In the using code on the right, the effectful computation in `g` escapes into the newly allocated data structure denoted by `f`. So `f` has type `Fun[int,void,H]`, assuming the handler is named `H`. But since `f` does not outlive the `try–with` that introduces the capability held by `H`, the code is safely accepted.

3.5 Implementation

This paper does not explore the options for implementing the new effect mechanism. However, implementation is largely an orthogonal concern. It appears entirely feasible to build on ongoing work on efficiently implementing algebraic effects [9, 28]. When algebraic effects are used as a termination-style exception mechanism, it is important that `try`-block computations be cheap; it should be possible to adapt the technique used by Zhang et al. [47], which corresponds to passing (static) capability labels rather than whole continuations.

4 A CORE LANGUAGE

To pin down the semantics of tunneled algebraic effects, we formally define a core calculus we call $\lambda_{⇩⇧}$, which captures the key aspects of the language mechanisms introduced in Section 3.

4.1 Syntax

The language $\lambda_{⇩⇧}$ is a simply typed lambda calculus, extended with language facilities essential to tunneling, including effect polymorphism, handler polymorphism, a way to access effect operations (⇧), and a way to discharge effects (⇩). For simplicity, it is assumed that handlers are always given explicitly for effectful computations (rather than resolving elided handlers to the closest lexically enclosing binding), that effect signatures contain exactly one effect operation, and that effect operations accept exactly one argument. Lifting these restrictions is straightforward, but adds syntactic complexity that obscures the key issues.

Like previous calculi, our formalism omits explicit handler state. But handler state can be encoded within the algebraic-effects framework—and consequently in $\lambda_{⇩⇧}$—as Bauer and Pretnar [5] show. It is also possible to extend the core calculus with handler state and, potentially, existentials to ensure encapsulation of the state. We expect such an extension to be largely orthogonal.

Figure 8 presents the syntax of $\lambda_{⇩⇧}$. An overline denotes a (possibly empty) sequence of syntactic objects. For instance, $\overline{e}$ denotes a list of effects, with an empty sequence denoted by $\varnothing$. The i-th element in a sequence $\overline{\bullet}$ is denoted by $\bullet^{(i)}$. Metavariables standing for identifiers are given a lighter color.

Types. Types include the base type $\mathbb{1}$, function types $S \to [T]_{\overline{e}}$, effect-polymorphic types $\forall \alpha.\, T$, and handler-polymorphic types $\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}$. The result type of a function type or that of a handler-polymorphic type can be annotated by effects. For brevity, we omit explicit annotations when there is no effect; for example, the type $S \to T$ means $S \to [T]_{\varnothing}$. Computations directly quantified by effect variables must be pure, an easily lifted simplification that matches both typical usage and previous formalizations (e.g., [7, 28]). Abstract handlers h implement effect signatures, whose names are ranged over by $\mathbb{F}$. We assume a global mapping from effect names to effect signatures; given an effect name $\mathbb{F}$, the helper function $op(\cdot)$ returns the type of its effect operation.

Terms. Terms consist of the standard ones of the simply typed lambda calculus plus those concerned with effects, including the ⇧- and ⇩-terms, effect-polymorphic abstraction $\Lambda \alpha.\, t$ and its application,

capability effects	e	::=	$\alpha \mid \ell \mid \mathsf{h}.\mathbf{lbl}$
types	T, S	::=	$\mathbb{1} \mid S \rightarrow [T]_{\overline{e}} \mid \forall \alpha.\, T \mid \Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}$
handlers	h, g	::=	$\mathsf{h} \mid H^{\ell}$
terms	t, s	::=	$() \mid \mathsf{x} \mid \lambda \mathsf{x}{:}T.\, t \mid t\ s \mid \mathbf{let}\ \mathsf{x}{:}T = t\ \mathbf{in}\ s \mid$
			$\Lambda\alpha.\, t \mid t\ [\overline{e}] \mid \lambda \mathsf{h}{:}\mathbb{F}.\, t \mid t\ h \mid ⇧ h \mid ⇩^{\ell}_{[T]_{\overline{e}}}\ t$
handler definitions	H, G	::=	$\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\, t$
effect var. environments	Δ	::=	$\varnothing \mid \Delta, \alpha$
handler var. environments	P	::=	$\varnothing \mid \mathrm{P}, \mathsf{h}{:}\mathbb{F}$
term var. environments	Γ	::=	$\varnothing \mid \Gamma, \mathsf{x}{:}T$
label environments	Ξ	::=	$\varnothing \mid \Xi, \ell{:}[T]_{\overline{e}}$

effect names $\mathbb{F}$ *labels* ℓ *effect variables* α *handler variables* h *term variables* $\mathsf{x}, \mathsf{y}, \mathsf{k}, \ldots$

Figure 8. Syntax of $\lambda_{⇩⇧}$

and handler-polymorphic abstraction $\lambda \mathsf{h}{:}\mathbb{F}.\, t$ and its application. The ⇧- and ⇩- terms, which we read as "up" and "down", correspond in the language of Section 3 to effect operations and effect handling.

For example, given a handler variable h that implements an effect $\mathbb{F}$ with signature $T_1 \rightarrow T_2$, the term $⇧\mathsf{h}$ is an effect operation whose implementation is provided by h, while the term $⇧\mathsf{h}\ v$ invokes the effect operation (assuming the value v has type T_1), raising an effect.

The `try-with` construct corresponds to terms of form

$$⇩^{\ell}_{[T]_{\overline{e}}}\ (\lambda \mathsf{h}{:}\mathbb{F}.\, t)\ \ H^{\ell}$$

where the term t corresponds to the computation in the `try` block, and H the handler in the `with` clause. Term t is placed in a handler-polymorphic abstraction, which is then immediately applied to the handler. The handler variable h, occurring free in t, can be thought of as creating a local binding for handler H that t uses to handle its effects.

As discussed in Section 3.4, a `try-with` expression implicitly marks a program point, creating a stack-region capability that is in scope within the `try` block. Correspondingly, ⇩-terms in $\lambda_{⇩⇧}$ mark program points that create capabilities. These capabilities are represented by labels ℓ; terms of form $⇩^{\ell}_{[T]_{\overline{e}}}\ t$ bind a label ℓ whose scope is t. Subterms of t can then use ℓ to show they possess the region capability. Labels bound by different ⇩-terms are assumed to be unique. To ensure unique typing, a ⇩-term is annotated with the type and effects $[T]_{\overline{e}}$ of the very term; they correspond to the type and effects of a `try-with` expression as a whole. We omit these annotations when they are irrelevant in the context.

To handle an effect requires both the handling code and the capability. Hence, handler definitions H are always tagged by a label in scope, forming pairs of form H^{ℓ}. Our use of ⇩-terms supports pairing different handler definitions with the same program point, a useful feature that is common in programming languages with exception handlers but that does not seem to be captured by previous formalisms. For example, the following term corresponds to associating two handlers with the same `try` block:

$$⇩^{\ell}_{[T]_{\overline{e}}}\left(\lambda \mathsf{h}_1{:}\mathbb{F}_1.\, (\lambda \mathsf{h}_2{:}\mathbb{F}_2.\, t)\ \ H_2^{\ell}\right)\ \ H_1^{\ell}$$

Handlers. A handler h is either a handler variable h or a definition–label pair H^{ℓ}. The (statically unknown) label embodied in a handler variable h is denoted by $\mathsf{h}.\mathbf{lbl}$. Substituting a handler of form H^{ℓ} for a handler variable h also replaces any occurrences of $\mathsf{h}.\mathbf{lbl}$ with ℓ.

Handler definitions H are of form $\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\, t$, where $\mathbb{F}$ is the effect signature being implemented and t is the handling code. Variables x and k may occur free in t: x denotes the argument passed to the effect operation, and k the continuation at the point the effect operation is invoked.

$$\begin{array}{lrcl}\text{values} & v, u & ::= & () \mid \lambda x\!:\!T.\,t \mid \Lambda\alpha.\,t \mid \lambda h\!:\!\mathbb{F}.\,t \mid \Uparrow H^{\ell} \\ \text{evaluation contexts} & K & ::= & [\cdot] \mid K\,t \mid v\,K \mid K\,[\overline{\ell}] \mid K\,H^{\ell} \mid \textbf{let}\ x\!:\!T = K\ \textbf{in}\ t \mid \Downarrow^{\ell} K\end{array}$$

$$\boxed{t_1 \longrightarrow t_2}$$

$$[\text{E-KTX}]\ \frac{t_1 \longrightarrow t_2}{K[t_1] \longrightarrow K[t_2]} \qquad [\text{E-APP}]\ (\lambda x\!:\!T.\,t)\ v \longrightarrow t\,\{v/x\} \qquad [\text{E-EAPP}]\ (\Lambda\alpha.\,t)\ [\overline{\ell}] \longrightarrow t\left\{\overline{\ell}\middle/\alpha\right\}$$

$$[\text{E-HAPP}]\quad (\lambda h\!:\!\mathbb{F}.\,t)\ H^{\ell} \longrightarrow t\left\{H^{\ell}\middle/h\right\} \qquad [\text{E-LET}]\quad \textbf{let}\ x\!:\!T = v\ \textbf{in}\ t \longrightarrow t\,\{v/x\}$$

$$[\text{E-DOWN-VAL}]\quad \Downarrow^{\ell} v \longrightarrow v \qquad [\text{E-DOWN-UP}]\quad \frac{H = \textbf{handler}^{\mathbb{F}}\ x\ k.\,t \qquad op(\mathbb{F}) = T_1 \rightarrow T_2}{\Downarrow^{\ell} K\left[\Uparrow H^{\ell}\ v\right] \longrightarrow t\left\{\lambda y\!:\!T_2.\,\Downarrow^{\ell} K[y]\middle/k\right\}\{v/x\}}$$

Figure 9. Operational semantics of $\lambda_{\Downarrow\Uparrow}$

Effects. The type system needs to track region capabilities as computational effects. An effect e is either an effect variable α, a label ℓ bound by a $\Downarrow$-term, or the label of a handler variable. With effects being just capabilities, we can handle effect composition simply: effect sequences $\overline{e}$ are essentially sets—the order and multiplicity of effects in a sequence are irrelevant. Substituting an effect sequence $\overline{e}$ for an effect variable α that is part of another effect sequence works by flattening $\overline{e}$ and replacing α with the flattened effects.

4.2 Operational Semantics

A small-step operational semantics of the core language is given in Figure 9. The semantics is defined in a largely standard way using evaluation contexts [18] with capture-avoiding substitution denoted by $\cdot\,\{\cdot/\cdot\}$. The transitive closure and the transitive, reflexive closure of the small-step transition relation $\longrightarrow$ are denoted by $\longrightarrow^{+}$ and $\longrightarrow^{*}$, respectively.

Of all the evaluation rules, [E-DOWN-UP] is most interesting, as it deals with the invocation of effect operations. Evaluating an invocation $\Uparrow H^{\ell}\ v$ amounts to evaluating the handling code in H, which requires the capability to access the stack regions marked by ℓ. Therefore, to reduce $\Uparrow H^{\ell}\ v$, the dynamic scope is searched for an evaluation context $\Downarrow^{\ell} K[\cdot]$ that binds ℓ. Notice that since labels bound by $\Downarrow$-terms are assumed to be unique, the inner context K does not further nest any evaluation context $\Downarrow^{\ell}[\cdot]$ binding the same label. This evaluation context K is then passed to the handling code as the resumption continuation. In case the handler chooses to abort the computation in K, evaluation continues with the surrounding evaluation context, as rule [E-KTX] suggests. Notice that K is guarded by $\Downarrow^{\ell}$ when passed to the handling code, so any invocation of effect operations labeled by ℓ in the resumption continuation can be handled properly.

4.3 Static Semantics

Some of the static-semantics rules of $\lambda_{\Downarrow\Uparrow}$ are provided in Figure 10. Term well-formedness rules have form $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}$, where Δ, P, Γ and Ξ are environments of free effect variables, handler variables, term variables, and labels, respectively. The judgment form says that under these environments the term t has type T and effects $\overline{e}$.

Rule [T-UP] suggests that an effect operation $\Uparrow h$ is a first-class value with type $T \rightarrow [S]_{e}$, where $T \rightarrow S$ is the effect signature and e is the capability held by h.

Rule [T-DOWN] suggests that a term t guarded by $\Downarrow^{\ell}$ possesses the capability ℓ: in the premise, t is typed under the label environment augmented with ℓ. Importantly, however, the label ℓ must not

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}}$$

$$[\textsc{t-up}]\quad \frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e \qquad op(\mathbb{F}) = T \to S}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Uparrow h : [T \to [S]_e]_\varnothing} \qquad [\textsc{t-down}]\quad \frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi, \ell : [T]_{\overline{e}} \vdash t : [T]_{\overline{e},\ell} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Downarrow^{\ell}_{[T]_{\overline{e}}} t : [T]_{\overline{e}}}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e}$$

$$[\textsc{t-hvar}]\quad \frac{\mathrm{P}(\mathsf{h}) = \mathbb{F}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{h} : \mathbb{F} \mid \mathsf{h}.\mathbf{lbl}} \qquad [\textsc{t-hdef}]\quad \frac{\Xi(\ell) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2 \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x} : T_1, \mathsf{k} : T_2 \to [S]_{\overline{e}} \mid \Xi \vdash t : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t\right)^{\ell} : \mathbb{F} \mid \ell}$$

Figure 10. Selected rules from the static semantics of $\lambda_{\Downarrow\Uparrow}$

occur free in the result type T and effects $\overline{e}$. Otherwise, ℓ could outlive its binding scope. For instance, it would then be possible to type the term $\Downarrow^{\ell}_{[S_1 \to [S_2]_\ell]_\varnothing} \left(\Uparrow H^{\ell}\right)$ as $S_1 \to [S_2]_\ell$, assuming H implements effect signature $S_1 \to S_2$. Per evaluation rule [E-DOWN-VAL], the term would then evaluate to $\Uparrow H^{\ell}$. But without a corresponding $\Downarrow^{\ell}$ in the dynamic context, an invocation of the effect operation $\Uparrow H^{\ell}\ t$ would get stuck.

Handler well-formedness rules have form $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e$, which states that handler h implements the algebraic effect $\mathbb{F}$ and has label e. Rule [T-HDEF] requires that the handling code t of a handler H^{ℓ} be typable using the type and effects $[S]_{\overline{e}}$ prescribed by the label ℓ. This requirement helps the reduction rule [E-DOWN-UP] preserve typing.

The other static semantics rules are largely standard and can be found in the appendices. These include the remaining rules for term well-formedness, the rules for the well-formedness of types and effects, and the rules for the partial orderings on types and effect sequences.

Encoding data structures. For simplicity, $\lambda_{\Downarrow\Uparrow}$ does not have data structures. However, $\lambda_{\Downarrow\Uparrow}$ allows their encoding via closures, where the captured variables may have latent polymorphic effects. For example, a simplified *pair* data structure polymorphic over the latent effects of its components can be encoded as follows:

$$\begin{array}{rcll}
T & \stackrel{def}{=} & S_1 \to [S_2]_\alpha & \textit{S_1 and S_2 can be any closed type} \\
\textit{pair} & \stackrel{def}{=} & \Lambda\alpha.\ \lambda\mathsf{x}:T.\ \lambda\mathsf{y}:T.\ \lambda\mathsf{f}:T \to T \to T.\ \mathsf{f}\ \mathsf{x}\ \mathsf{y} & \textit{construct a pair} \\
\textit{first} & \stackrel{def}{=} & \Lambda\alpha.\ \lambda\mathsf{p}:(T \to T \to T) \to T.\ \mathsf{p}\ \ (\lambda\mathsf{x}:T.\ \lambda\mathsf{y}:T.\ \mathsf{x}) & \textit{obtain the first component} \\
\textit{second} & \stackrel{def}{=} & \Lambda\alpha.\ \lambda\mathsf{p}:(T \to T \to T) \to T.\ \mathsf{p}\ \ (\lambda\mathsf{x}:T.\ \lambda\mathsf{y}:T.\ \mathsf{y}) & \textit{obtain the second component}
\end{array}$$

The two components, both having type T, have α as their latent effects. The *pair* constructor is then polymorphic in α.

This example cannot be readily encoded in previous formalisms [36, 47], which support a limited form of effect polymorphism by introducing second-class values that cannot escape their defining scope. In particular, these systems do not admit the subterm $\lambda\mathsf{x} : T.\ \lambda\mathsf{y} : T.\ \mathsf{x}$ in the definition of *first*, or the subterm $\lambda\mathsf{y} : T.\ \mathsf{y}$ in the definition of *second*. Variable x in the first subterm, being second-class because it has a polymorphic latent effect, escapes its defining scope via the closure $\lambda\mathsf{y} : T.\ \mathsf{x}$ capturing it. Similarly, in the second subterm, variable y escapes its defining scope. By contrast, our use of explicit effect polymorphism and capability labels enables the definition of effect-polymorphic data structures.

$$
\begin{aligned}
C \quad ::= \quad & [\cdot] \mid C[\lambda \mathsf{x}\!:\!T.\,[\cdot]] \mid C[[\cdot]\ t] \mid C[t\ [\cdot]] \mid C[\textbf{let}\ \mathsf{x}\!:\!T = [\cdot]\ \textbf{in}\ t] \mid \\
& C[\textbf{let}\ \mathsf{x}\!:\!T = t\ \textbf{in}\ [\cdot]] \mid C[\Lambda\alpha.\,[\cdot]] \mid C[[\cdot]\ \lfloor\overline{e}\rfloor] \mid C[\lambda \mathsf{h}\!:\!\mathbb{F}.\,[\cdot]] \mid C[[\cdot]\ h] \mid \\
& C\Big[t\ \big(\textbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\,[\cdot]\big)^{\ell}\Big] \mid C\Big[\Uparrow \big(\textbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\,[\cdot]\big)^{\ell}\Big] \mid C\big[\Downarrow^{\ell}\ [\cdot]\big]
\end{aligned}
$$

Figure 11. Program contexts of $\lambda_{\Downarrow\Uparrow}$

4.4 Contextual Refinement and Equivalence

A *program context* is a program with a hole $[\cdot]$ in it. Figure 11 shows the different types of program contexts in $\lambda_{\Downarrow\Uparrow}$. Well-formedness judgments for program contexts have the form

$$\vdash C \;:\; \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T$$

The meaning of this judgment is that if a term t satisfies the typing judgment $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [S]_{\overline{e}}$, then plugging t into C results in a program that satisfies $\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash C[t] : [T]_{\varnothing}$. These rules are available in the appendices.

Our goal is to prove that with tunneling, algebraic effects can preserve abstraction. Abstraction is shown by demonstrating that implementations using effects internally cannot be distinguished by external observers. The gold standard of indistinguishability is *contextual equivalence*: two terms are contextually equivalent if plugging them into an arbitrary well-formed program context always gives two programs whose evaluations yield the same observation [33].

We define contextual equivalence in terms of *contextual refinement*, a weaker, asymmetric relation that requires one term to be able to simulate the behaviors of the other:

Definition 1 (contextual refinement $\preccurlyeq_{ctx}$ and contextual equivalence $\approx_{ctx}$).

$$
\begin{aligned}
\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{ctx} t_2 : [T]_{\overline{e}} \quad & \stackrel{def}{=} \quad \forall C.\ \vdash C \;:\; \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \Rightarrow \\
& \qquad (\exists v_1.\, C[t_1] \longrightarrow^* v_1) \Rightarrow (\exists v_2.\, C[t_2] \longrightarrow^* v_2) \\
\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \approx_{ctx} t_2 : [T]_{\overline{e}} \quad & \stackrel{def}{=} \quad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{ctx} t_2 : [T]_{\overline{e}} \ \wedge\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_2 \preccurlyeq_{ctx} t_1 : [T]_{\overline{e}}
\end{aligned}
$$

For programs to be equivalent in the above definition, they only need to agree on termination, but this seemingly weak observation of program behavior does not weaken the discriminating power of the definition, because of the universal quantification over all possible program contexts and because $\lambda_{\Downarrow\Uparrow}$ is Turing-complete (see Section 5.1). Hence, if two computations that reduce to observably different values, one can always construct a program context that makes the two computations exhibit different termination behavior.

However, the universal quantification over contexts also makes it hard to show equivalence by using the definition directly. We therefore take one of the standard approaches to establishing contextual equivalence: constructing a logical relation that implies contextual equivalence.

5 A SOUND LOGICAL-RELATIONS MODEL

We develop a logical-relations model for $\lambda_{\Downarrow\Uparrow}$ and prove the important property that logically related terms are contextually equivalent. This semantic soundness result guarantees that the language $\lambda_{\Downarrow\Uparrow}$ is both type-safe and abstraction-safe.

5.1 Step Indexing

A logical-relations model gives a relational interpretation of types, traditionally defined inductively on the structure of types. But language features like recursive types require a more sophisticated induction principle. Algebraic effects present a similar challenge because effect signatures can be defined recursively.

Recursively defined effect signatures give rise to programs that diverge, and consequently make the language Turing-complete. For example, suppose effect $\mathbb{F}$ has signature $op(\mathbb{F}) = \mathbb{1} \to \Pi_{\mathsf{h}:\mathbb{F}} [T]_{\mathsf{h}.\mathbf{lbl}}$, which recursively mentions $\mathbb{F}$, and that H is defined as follows:

$$H \stackrel{def}{=} \mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ \mathsf{k}\ \ (\lambda \mathsf{h}:\mathbb{F}.\Uparrow \mathsf{h}\ ()\ \mathsf{h})$$

Then the evaluation of the program $\Downarrow^{\ell}_{[T]_{\varnothing}} (\lambda \mathsf{h}:\mathbb{F}.\Uparrow \mathsf{h}\ ()\ \mathsf{h})\ \ H^{\ell}$ does not terminate:

$$\begin{aligned}&\Downarrow^{\ell} (\lambda \mathsf{h}:\mathbb{F}.\Uparrow \mathsf{h}\ ()\ \mathsf{h})\ \ H^{\ell} \longrightarrow \Downarrow^{\ell} \left(\Uparrow H^{\ell}\ ()\ H^{\ell}\right) \longrightarrow \left(\lambda \mathsf{y}:\Pi_{\mathsf{h}:\mathbb{F}} [T]_{\mathsf{h}.\mathbf{lbl}}.\Downarrow^{\ell} \mathsf{y}\ H^{\ell}\right)\ \ (\lambda \mathsf{h}:\mathbb{F}.\Uparrow \mathsf{h}\ ()\ \mathsf{h}) \\ \longrightarrow\ &\Downarrow^{\ell} (\lambda \mathsf{h}:\mathbb{F}.\Uparrow \mathsf{h}\ ()\ \mathsf{h})\ \ H^{\ell} \longrightarrow \cdots\end{aligned}$$

Because of this recursion in the signature of $\mathbb{F}$, structural induction alone is unable to give a well-defined relational interpretation of $\mathbb{F}$.

$$[\text{LÖB}]\quad \frac{P, \triangleright Q \vdash Q}{P \vdash Q} \qquad\qquad [\text{MONO}]\quad \frac{P, Q \vdash R}{P, \triangleright Q \vdash \triangleright R}$$

Figure 12. Rules for $\triangleright$

Step indexing [2] has been successfully applied to cope with recursive types (e.g., by Ahmed [1]). In this approach, the logical relation is defined using a double induction, first on a step index, and second on the structure of types. Intuitively, the step index indicates for how many evaluation steps the proposition is true; at step 0 everything is vacuously true, and if a proposition is true for any number of steps then it is true in a non-step-indexed setting.

Our definition is step-indexed. It uses a logic equipped with the modality $\triangleright$, read as "later", which offers a clean abstraction of step indexing [3, 15]. If proposition P holds for n steps, then $\triangleright P$ holds for $n-1$ steps. So P implies $\triangleright P$. Importantly, the $\triangleright$ modality provides the [LÖB] axiom (Figure 12), which can be viewed as an induction principle on step indices. The $\triangleright$ modality distributes over other connectives, so rule [MONO] is derivable.

As we shall see in Section 5.3, to ensure well-definedness, recursive invocations of the interpretation of effect signatures occur under the $\triangleright$ modality.

5.2 A Biorthogonal Term Relation

We introduce a logical relation for terms, which are closed under the empty variable environments but may use capability labels that are not locally bound. The term relation is defined using the technique of *biorthogonality*, pioneered by Pitts and Stark [37]. Biorthogonality, also known as $\top\top$-closure, lends itself to languages whose operational semantics manipulate evaluation contexts [7, 16, 24]: in a biorthogonal term relation, two terms are related if evaluating them in related evaluation contexts yields related observations. Hence, our term relation $\mathcal{T}$ is defined as follows, with a relation $\mathcal{K}$ providing a notion of relatedness for evaluation contexts and relation $\mathcal{O}$ relating observations:

$$\begin{aligned}
\mathcal{O}\,(t_1,\ t_2) &\stackrel{def}{=} (\exists v_1, v_2.\ t_1 = v_1\ \wedge\ t_2 \longrightarrow^* v_2)\ \vee\ \left(\exists t_1'.\ t_1 \longrightarrow t_1'\ \wedge\ \triangleright \mathcal{O}\left(t_1',\ t_2\right)\right) \\
\mathcal{T}[\![[T]_{\overline{\varepsilon}}]\!]^{\rho}_{\delta}\,(t_1,\ t_2) &\stackrel{def}{=} \forall K_1, K_2.\ \mathcal{K}[\![[T]_{\overline{\varepsilon}}]\!]^{\rho}_{\delta}\,(K_1,\ K_2) \Rightarrow \mathcal{O}\,(K_1[t_1],\ K_2[t_2]) \\
\mathcal{K}[\![[T]_{\overline{\varepsilon}}]\!]^{\rho}_{\delta}\,(K_1,\ K_2) &\stackrel{def}{=} \left(\forall v_1, v_2.\ \mathcal{V}[\![T]\!]^{\rho}_{\delta}\,(v_1,\ v_2) \Rightarrow \mathcal{O}\,(K_1[v_1],\ K_2[v_2])\right)\ \wedge \\
&\qquad \left(\forall t_1, t_2.\ \mathcal{S}[\![[T]_{\overline{\varepsilon}}]\!]^{\rho}_{\delta}\,(t_1,\ t_2) \Rightarrow \mathcal{O}\,(K_1[t_1],\ K_2[t_2])\right) \\
\mathcal{S}[\![[T]_{\overline{\varepsilon}}]\!]^{\rho}_{\delta}\,(K_1[t_1],\ K_2[t_2]) &\stackrel{def}{=} \exists \psi, \overline{\ell_1}, \overline{\ell_2}.\ \mathcal{U}[\![\overline{\varepsilon}]\!]^{\rho}_{\delta}\left(t_1,\ t_2,\ \psi,\ \overline{\ell_1},\ \overline{\ell_2}\right)\ \wedge \\
&\qquad \left(\forall i.\ \ell_1^{(i)} \curvearrowright K_1\right)\ \wedge\ \left(\forall i.\ \ell_2^{(i)} \curvearrowright K_2\right)\ \wedge \\
&\qquad \forall t_1', t_2'.\ \psi\left(t_1',\ t_2'\right) \Rightarrow \triangleright \mathcal{T}[\![[T]_{\overline{\varepsilon}}]\!]^{\rho}_{\delta}\left(K_1\left[t_1'\right],\ K_2\left[t_2'\right]\right)
\end{aligned}$$

Apart from the $\mathcal{S}$ relation, the above definitions are standard. We define logical equivalence in terms of a notion of *logical refinement*, in much the same way that we define contextual equivalence in terms of contextual refinement. Rather than requiring the terms to exhibit the same termination behavior, the observation relation $\mathcal{O}$ relates two computations where termination of the first computation merely *implies* that of the second one. The $\mathcal{O}$ relation is defined recursively; the use of the $\triangleright$ modality

$$\boxed{\ell \curvearrowright K} \quad \ell \curvearrowright [\cdot] \quad \frac{\ell \curvearrowright K}{\ell \curvearrowright K\; t} \quad \frac{\ell \curvearrowright K}{\ell \curvearrowright v\; K} \quad \frac{\ell \curvearrowright K}{\ell \curvearrowright K\;[\overline{\ell}]} \quad \frac{\ell \curvearrowright K}{\ell \curvearrowright K\; H^{\ell}} \quad \frac{\ell \curvearrowright K}{\ell \curvearrowright \mathbf{let}\; x{:}T = K \;\mathbf{in}\; t} \quad \frac{\ell_1 \curvearrowright K \qquad \ell_1 \neq \ell_2}{\ell_1 \curvearrowright \Downarrow^{\ell_2} K}$$

Figure 13. $\ell \curvearrowright K$ means the evaluation context K does not bind label ℓ

suggests that the definition is implicitly indexed by the number of remaining evaluation steps the first computation can take.

Two evaluation contexts are related by $\mathcal{K}$ if they yield related observations when applied to related values. However, in the presence of algebraic effects, values are not the only kind of irreducible term. Terms of form $K\left[\Uparrow H^{\ell}\; v\right]$ where the evaluation context K does not bind ℓ (captured by the judgment form $\ell \curvearrowright K$ defined in Figure 13) are stuck when put into an empty evaluation context.

So we borrow from Biernacki et al. [7] a logical relation $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, which, being a smaller relation than $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, relates two computations that can possibly get stuck by themselves because they raise effects among $\overline{e}$. The definition of the $\mathcal{K}$ relation then requires that two related evaluation contexts yield related observations when applied to not only values related by $\mathcal{V}$ but also terms related by $\mathcal{S}$. The $\mathcal{S}$ relation is discussed further in Section 5.3.

Because of the use of biorthogonality, and assuming parametricity is derivable, our term relation is automatically complete with respect to contextual refinement [16, 37]: contextually equivalent terms are always logically related. So the key theorems to prove are parametricity and soundness.

The definitions of the relations $\mathcal{T}$, $\mathcal{K}$, and $\mathcal{S}$ are mutually recursive, and are dependent on the semantic interpretation of a type $\mathcal{V}[\![T]\!]^{\rho}_{\delta}$ and that of an effect sequence $\mathcal{U}[\![\overline{e}]\!]^{\rho}_{\delta}$, defined below.

5.3 Semantic Types, Semantic Effect Signatures, and Semantic Effects

The logical relation $\mathcal{V}[\![T]\!]^{\rho}_{\delta}$ (Figure 14), defined by structural induction on the type T, interprets T as a binary relation on values. The unit type and function types are interpreted in a standard way, following the contract that the logical relation should be preserved by the elimination (or introduction) forms of the types.

Effect-polymorphic types and handler-polymorphic types bind effect variables and handler variables. Accordingly, environments δ and ρ are introduced to provide substitutions for variables occurring free in the type being interpreted:

$$\delta ::= \varnothing \;\mid\; \delta,\, \alpha \mapsto \left\langle \overline{\ell_1}, \overline{\ell_2}, \phi \right\rangle \qquad \rho ::= \varnothing \;\mid\; \rho,\, \mathsf{h} \mapsto \left\langle H_1^{\ell_1}, H_2^{\ell_2}, \eta \right\rangle$$

We use δ_1 and δ_2 (resp. ρ_1 and ρ_2) to mean the substitution functions for free effect (resp. handler) variables. In addition to these syntactic substitution functions, the environment δ maps each effect variable to a third component that is the semantic interpretation chosen for the effect variable, while the environment ρ maps each handler variable to a third component that is the term relation the computations of the two handlers satisfy. (Metavariables ϕ, η, and ψ range over relation variables.) The definitions in Figure 14 are also parameterized by a label environment Ξ; labels in the domain of Ξ may occur free in the types and effects being interpreted. We omit Ξ for brevity.

The definition of $\mathcal{V}[\![\forall \alpha.\, T]\!]^{\rho}_{\delta}$ shows the source of the abstraction guarantees provided by effect-polymorphic abstractions: two effect-polymorphic abstractions are related if their applications are related however the effect variable is interpreted. The definition of $\mathcal{V}[\![\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}]\!]^{\rho}_{\delta}$ says that two handler-polymorphic abstractions are related if their applications to any related handlers are related. Handler-relatedness is defined by the logical relation $\mathcal{H}[\![\mathbb{F}]\!]$, indexed by effect signatures $\mathbb{F}$. As discussed in Section 5.1, effect signatures can be recursively defined. Thus $\mathcal{H}[\![\mathbb{F}]\!]$ is invoked here under the $\triangleright$ modality so that the definition is admissible.

The interpretation of an effect signature $\mathbb{F}$ is similar to that of a function type: two handlers are related if their handling code is related under any related substitutions for the free variables. $\mathcal{H}[\![\mathbb{F}]\!]$

Semantic types:

$$\begin{aligned}
\mathcal{V}[\![\mathbb{1}]\!]^{\rho}_{\delta}(v_1, v_2) &\overset{def}{=} v_1 = () \ \wedge\ v_2 = () \\
\mathcal{V}[\![T \rightarrow [S]_{\overline{e}}]\!]^{\rho}_{\delta}(v_1, v_2) &\overset{def}{=} \forall u_1, u_2.\ \mathcal{V}[\![T]\!]^{\rho}_{\delta}(u_1, u_2) \Rightarrow \mathcal{T}[\![[S]_{\overline{e}}]\!]^{\rho}_{\delta}(v_1\ u_1,\ v_2\ u_2) \\
\mathcal{V}[\![\forall\alpha.\ T]\!]^{\rho}_{\delta}(v_1, v_2) &\overset{def}{=} \forall \overline{\ell_1}, \overline{\ell_2}, \phi.\ \mathcal{T}[\![[T]_{\varnothing}]\!]^{\rho}_{\delta,\ \alpha\mapsto\langle\overline{\ell_1},\, \overline{\ell_2},\, \phi\rangle}\left(v_1\ [\overline{\ell_1}],\ v_2\ [\overline{\ell_2}]\right) \\
\mathcal{V}[\![\Pi_{\mathsf{h}:\mathbb{F}}\ [T]_{\overline{e}}]\!]^{\rho}_{\delta}(v_1, v_2) &\overset{def}{=} \forall H_1^{\ell_1}, H_2^{\ell_2}, \eta.\ \triangleright\mathcal{H}[\![\mathbb{F}]\!]\left(H_1^{\ell_1},\ H_2^{\ell_2},\ \eta\right) \Rightarrow \\
&\qquad \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho,\ \mathsf{h}\mapsto\langle H_1^{\ell_1},\, H_2^{\ell_2},\, \eta\rangle}_{\delta}\left(v_1\ H_1^{\ell_1},\ v_2\ H_2^{\ell_2}\right)
\end{aligned}$$

Semantic effect signatures:

$$\begin{aligned}
\mathcal{H}[\![\mathbb{F}]\!]\left(H_1^{\ell_1},\ H_2^{\ell_2},\ \eta\right) &\overset{def}{=} H_i = \textbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_i\ (i = 1, 2) \ \wedge\ op(\mathbb{F}) = T_1 \rightarrow T_2 \ \wedge \\
&\quad \forall v_1, v_2.\ \mathcal{V}[\![T_1]\!]^{\varnothing}_{\varnothing}(v_1, v_2) \Rightarrow \\
&\quad \forall u_1, u_2.\ \left(\forall w_1, w_2.\ \mathcal{V}[\![T_2]\!]^{\varnothing}_{\varnothing}(w_1, w_2) \Rightarrow \eta\,(u_1\ w_1,\ u_2\ w_2)\right) \Rightarrow \\
&\quad \eta\,(t_1\,\{u_1/\mathsf{k}\}\,\{v_1/\mathsf{x}\},\ t_2\,\{u_2/\mathsf{k}\}\,\{v_2/\mathsf{x}\})
\end{aligned}$$

Semantic effects:

$$\begin{aligned}
\mathcal{U}[\![\alpha]\!]^{\rho}_{\delta}\left(t_1,\ t_2,\ \psi,\ \overline{\ell_1},\ \overline{\ell_2}\right) &\overset{def}{=} \delta(\alpha) = \left\langle\overline{\ell_1'}, \overline{\ell_2'}, \phi\right\rangle \ \wedge\ \phi\left(t_1,\ t_2,\ \psi,\ \overline{\ell_1},\ \overline{\ell_2}\right) \\
\mathcal{U}[\![e]\!]^{\rho}_{\delta}(t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2) &\overset{def}{=} \rho_1 e = \ell_1 \ \wedge\ \rho_2 e = \ell_2 \ \wedge \\
&\quad \left(\mathcal{U}_A[\![e]\!]^{\rho}_{\delta}(t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2) \ \vee\ \mathcal{U}_B[\![e]\!](t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2)\right) \\
\mathcal{U}_A[\![e]\!]^{\rho}_{\delta}(t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2) &\overset{def}{=} t_1 = \Uparrow H_1^{\ell_1}\ v_1 \ \wedge\ t_2 = \Uparrow H_2^{\ell_2}\ v_2 \ \wedge\ \triangleright\mathcal{H}[\![\mathbb{F}]\!]\left(H_1^{\ell_1},\ H_2^{\ell_2},\ \mathcal{W}[\![e]\!]^{\rho}_{\delta}\right) \ \wedge \\
&\quad op(\mathbb{F}) = T \rightarrow T' \ \wedge\ \triangleright\mathcal{V}[\![T]\!]^{\varnothing}_{\varnothing}(v_1, v_2) \ \wedge\ \psi \equiv \triangleright\mathcal{V}[\![T']\!]^{\varnothing}_{\varnothing} \\
\mathcal{U}_B[\![e]\!](t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2) &\overset{def}{=} \left(\forall K.\ \ell_1 \curvearrowright K \Rightarrow \Downarrow^{\ell_1} K[t_1] \longrightarrow^{+} \Downarrow^{\ell_1} K\left[t_1'\right]\right) \ \wedge \\
&\quad \left(\forall K.\ \ell_2 \curvearrowright K \Rightarrow \Downarrow^{\ell_2} K[t_2] \longrightarrow^{*} \Downarrow^{\ell_2} K\left[t_2'\right]\right) \ \wedge\ \psi \equiv \left\{\left(t_1',\ t_2'\right)\right\} \\
\mathcal{U}[\![\overline{e}]\!]^{\rho}_{\delta}(t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2) &\overset{def}{=} \exists i.\ \mathcal{U}\left[\!\!\left[e^{(i)}\right]\!\!\right]^{\rho}_{\delta}(t_1,\ t_2,\ \psi,\ \ell_1,\ \ell_2)
\end{aligned}$$

Semantic labels:

$$\begin{aligned}
\mathcal{W}[\![\mathsf{h}.\textbf{lbl}]\!]^{\rho}_{\delta}(t_1,\ t_2) &\overset{def}{=} \rho(\mathsf{h}) = \left\langle H_1^{\ell_1}, H_2^{\ell_2}, \eta\right\rangle \ \wedge\ \eta\,(t_1,\ t_2) \\
\mathcal{W}[\![\ell]\!]^{\rho}_{\delta}(t_1,\ t_2) &\overset{def}{=} \Xi(\ell) = [T]_{\overline{e}} \ \wedge\ \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(t_1,\ t_2)
\end{aligned}$$

Figure 14. Relational interpretation of types, effect signatures, and effects

relates a third component η that is a term relation; the handler computations are in this relation. $\mathcal{H}[\![\mathbb{F}]\!]$ is not indexed by environments δ and ρ, because effect signatures are closed.

We revisit the definition of the $\mathcal{S}$ relation introduced in Section 5.2. As mentioned earlier, $\mathcal{S}$ can relate terms of form $K\left[\Uparrow H^{\ell}\ v\right]$ where $\ell \curvearrowright K$—although terms in this relation are not necessarily effectful, because it is possible for programs that use effects and those that do not to be equivalent. The operational meaning of these terms depends upon a larger surrounding context that binds the label ℓ. Therefore, the relation $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$ is defined using the $\mathcal{U}[\![\overline{e}]\!]^{\rho}_{\delta}$ relation, which relates the (possibly) effectful computations t_1 and t_2 and also a binary term relation $\psi \in \mathbb{P}(\mathsf{Term} \times \mathsf{Term})$ specifying the *outcomes* of these computations in a larger context. Given this specification, the definition of $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$ checks that plugging any pair of terms (t_1', t_2') related by the outcome specification into the current evaluation contexts yield related terms. Notice that $K_1\left[t_1'\right]$ and $K_2\left[t_2'\right]$ only need to be related in the future as indicated by the use of the $\triangleright$ modality, because it takes evaluation steps to reach t_1'.

Capability effects are interpreted by the $\mathcal{U}[\![e]\!]^{\rho}_{\delta}$ relation. For an effect variable α, the interpretation is simply the relation mapped to by the environment δ. For an effect of form ℓ or $\mathsf{h}.\textbf{lbl}$, two interpretations are provided. Relation $\mathcal{U}_A[\![e]\!]^{\rho}_{\delta}$ relates two effect operation invocations: $\Uparrow H_1^{\ell_1}\ v_1$ and $\Uparrow H_2^{\ell_2}\ v_2$ are related provided the handlers $H_1^{\ell_1}$ and $H_2^{\ell_2}$ are related and the arguments v_1 and v_2 are related. The outcome relation ψ in this case is the value relation at the return type of the effect operation. The interpretation of ℓ and that of $\mathsf{h}.\mathbf{lbl}$ differ in the relation that the handlers satisfy, captured by the two cases in the definition of $\mathcal{W}[\![e]\!]^{\rho}_{\delta}$: for $\mathsf{h}.\mathbf{lbl}$, this relation is the one that ρ maps h to, while for ℓ, this relation is $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, provided the label environment Ξ maps ℓ to $[T]_{\overline{e}}$. Relation $\mathcal{U}_B[\![e]\!]$ relates two terms t_1 and t_2 when evaluating them in evaluation contexts of form $\Downarrow^{\ell} K[\cdot]$ (where K does not bind ℓ) preserves the evaluation contexts.

The interpretation of a sequence of effects $\overline{e}$ is naturally the union of the interpretation of the individual effects in the sequence.

5.4 Properties of the Logical Relations

Basic properties. We point out some basic properties of the logical relations. These properties are employed by the proof leading to the soundness theorem and are used frequently in proofs of logical relatedness.

The following lemma applies when the goal is to prove the relatedness of two terms in which the subterms in the evaluation contexts are related:

Lemma 1. Given evaluation contexts K_1 and K_2, if
(a) for any v_1 and v_2, $\mathcal{V}[\![T]\!]^{\rho}_{\delta}(v_1,\, v_2)$ implies $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^{\rho}_{\delta}(K_1[v_1],\, K_2[v_2])$, and
(b) for any s_1 and s_2, $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(s_1,\, s_2)$ implies $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^{\rho}_{\delta}(K_1[s_1],\, K_2[s_2])$,
then for any t_1 and t_2, $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(t_1,\, t_2)$ implies $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^{\rho}_{\delta}(K_1[t_1],\, K_2[t_2])$.

The lemma says it suffices to show the evaluation contexts K_1 and K_2 satisfy the following conditions: applying K_1 and K_2 to (a) related values and (b) related terms in the $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$ relation yields related terms in the $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^{\rho}_{\delta}$ relation. We capture the preconditions of Lemma 1 by defining a logical relation $\mathcal{K}_{\mathcal{T}}[\![[T]_{\overline{e}} \rightsquigarrow [T']_{\overline{e'}}]\!]^{\rho}_{\delta}$: two evaluation contexts K_1 and K_2 are in this relation precisely when they satisfy the preconditions (a) and (b) of Lemma 1.

The following two lemmas show that reduction on either side reflects the term relation:

Lemma 2. If $t_1 \longrightarrow t_1'$ and $\triangleright\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\left(t_1',\, t_2\right)$, then $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(t_1,\, t_2)$.

Lemma 3. If $t_2 \longrightarrow t_2'$ and $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\left(t_1,\, t_2'\right)$, then $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(t_1,\, t_2)$.

The asymmetry with respect to the use of the $\triangleright$ modality in the preconditions is a result of the asymmetry in the definition of the $\mathcal{O}$ relation.

The following lemma allows proving two terms related by showing that they are in the $\mathcal{V}$ relation or in the $\mathcal{S}$ relation:

Lemma 4. $\mathcal{V}[\![T]\!]^{\rho}_{\delta} \subseteq \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta} \;\wedge\; \mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta} \subseteq \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$

These basic properties (Lemmas 1 to 4) are a consequence of the biorthogonal, step-indexed term relation defined in Section 5.2.

Soundness. Contextual refinement is defined for open terms, so we lift the term relation and the handler relation to open terms and open handlers by quantifying over related closing substitutions for the variable environments, as shown in Figure 15. Here, γ provides substitution functions for term variables: $\gamma ::= \varnothing \mid \gamma,\, \mathsf{x} \mapsto \langle v_1, v_2\rangle$. The interpretation of variable environments as relations on substitutions, also given in Figure 15, is standard.

Central to the proof of soundness are the compatibility lemmas; they show that logical refinement $\preccurlyeq_{log}$ is preserved by the syntactic typing rules. Figure 16 shows those compatibility lemmas corresponding to the typing rules in Figure 10, while the rest can be found in the appendices. Parametricity, and the fact that well-formed program contexts preserve logical refinement, are direct consequences of the compatibility lemmas:

$$\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}} \stackrel{def}{=} \forall \delta.\ [\![\Delta]\!]\,(\delta) \Rightarrow \forall \rho.\ [\![\mathrm{P}]\!]\,(\rho) \Rightarrow \forall \gamma.\ [\![\Gamma]\!]^{\rho}_{\delta}\,(\gamma) \Rightarrow \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\,(\delta_1\rho_1\gamma_1 t_1,\ \delta_2\rho_2\gamma_2 t_2)$$

$$\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid e \stackrel{def}{=} \forall \delta.\ [\![\Delta]\!]\,(\delta) \Rightarrow \forall \rho.\ [\![\mathrm{P}]\!]\,(\rho) \Rightarrow \forall \gamma.\ [\![\Gamma]\!]^{\rho}_{\delta}\,(\gamma) \Rightarrow \mathcal{H}[\![\mathbb{F}]\!]\left(\delta_1\rho_1\gamma_1 h_1,\ \delta_2\rho_2\gamma_2 h_2,\ \mathcal{W}[\![e]\!]^{\rho}_{\delta}\right)$$

$$[\![\varnothing]\!]\,(\delta) \stackrel{def}{=} \delta = \varnothing \qquad [\![\Delta, \alpha]\!]\,(\delta) \stackrel{def}{=} \delta = \delta',\ \alpha \mapsto \left\langle \overline{\ell_1}, \overline{\ell_2}, \phi \right\rangle \ \wedge\ [\![\Delta]\!]\,(\delta')$$

$$[\![\varnothing]\!]\,(\rho) \stackrel{def}{=} \rho = \varnothing \qquad [\![\mathrm{P}, \mathsf{h}{:}\mathbb{F}]\!]\,(\rho) \stackrel{def}{=} \rho = \rho',\ \mathsf{h} \mapsto \left\langle H_1^{\ell_1}, H_2^{\ell_2}, \eta \right\rangle \ \wedge\ [\![\mathrm{P}]\!]\,(\rho') \ \wedge\ \mathcal{H}[\![\mathbb{F}]\!]\left(H_1^{\ell_1},\ H_2^{\ell_2},\ \eta\right)$$

$$[\![\varnothing]\!]^{\rho}_{\delta}\,(\gamma) \stackrel{def}{=} \gamma = \varnothing \qquad [\![\Gamma, \mathsf{x}{:}T]\!]^{\rho}_{\delta}\,(\gamma) \stackrel{def}{=} \gamma = \gamma',\ \mathsf{x} \mapsto \langle v_1, v_2 \rangle \ \wedge\ [\![\Gamma]\!]^{\rho}_{\delta}\,(\gamma') \ \wedge\ \mathcal{V}[\![T]\!]^{\rho}_{\delta}\,(v_1,\ v_2)$$

Figure 15. Logical relations for open terms and handlers

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid e \qquad op(\mathbb{F}) = T \rightarrow S}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Uparrow h_1 \preccurlyeq_{log} \Uparrow h_2 : [T \rightarrow [S]_e]_{\varnothing}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi, \ell{:}[T]_{\overline{e}} \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}, \ell} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Downarrow^{\ell}_{[T]_{\overline{e}}} t_1 \preccurlyeq_{log} \Downarrow^{\ell}_{[T]_{\overline{e}}} t_2 : [T]_{\overline{e}}}$$

$$\frac{\mathrm{P}(\mathsf{h}) = \mathbb{F}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{h} \preccurlyeq_{log} \mathsf{h} : \mathbb{F} \mid \mathsf{h}.\mathbf{lbl}}$$

$$\frac{\Xi(\ell) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \rightarrow T_2 \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x}{:}T_1, \mathsf{k}{:}T_2 \rightarrow [S]_{\overline{e}} \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_1\right)^{\ell} \preccurlyeq_{log} \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_2\right)^{\ell} : \mathbb{F} \mid \ell}$$

Figure 16. Selected compatibility lemmas. The lemmas are written in the style of inference rules so that they can be read in tandem with the corresponding typing rules [T-UP], [T-DOWN], [T-HVAR], and [T-HDEF] in Figure 10.

Theorem 1 (Parametricity, a.k.a., Fundamental Property, a.k.a., Abstraction Theorem).

(1) $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}} \Rightarrow \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t \preccurlyeq_{log} t : [T]_{\overline{e}}$

(2) $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e \Rightarrow \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h \preccurlyeq_{log} h : \mathbb{F} \mid e$

Lemma 5 (Congruency). $\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \ \wedge\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}} \Rightarrow \varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash C[t_1] \preccurlyeq_{log} C[t_2] : [T']_{\varnothing}$

One last step leading to the soundness theorem is to show the logical relation is adequate—two logically related pure terms are observationally related:

Lemma 6 (Adequacy). $\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\varnothing} \Rightarrow \mathcal{O}\,(t_1,\ t_2)$

Type safety, the property that well-typed programs can only evaluate to values or diverge, falls out as an easy corollary of Adequacy and Parametricity, as the $\mathcal{O}$ relation only relates terms whose evaluations do not get stuck.

Theorem 2 (Type Safety). If $\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash t : [T]_{\varnothing}$ and $t \longrightarrow^{*} t'$, then either there exists v such that $t' = v$ or there exists t'' such that $t' \longrightarrow t''$.

The key theorem that logical refinement implies contextual refinement—and therefore logical equivalence implies contextual equivalence—is a result of Adequacy and Congruency:

Theorem 3 (Soundness). $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}} \Rightarrow \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{ctx} t_2 : [T]_{\overline{e}}$

5.5 Formalization in Coq

The definitions and results presented in Sections 5.3–5.4 have also been formalized using the Coq proof assistant [11]. The implementation consists of about 4,000 lines of code for defining the language and proving syntactic properties, and another 4,200 lines of code for defining the logical relations and proving their properties.

The logical relations are defined using the IxFree library [40], which is a shallow embedding of Dreyer et al.'s logic LSLR [15] in Coq. It also provides tactics for manipulating inference rules such as [LÖB] and [MONO], as well as a fixed-point operator for functions contractive in the use of the step index. Because IxFree does not support dependently typed fixpoint functions and because we use a dependently typed variant of de Bruijn indices, in our Coq formalization the type and effects attached to a label must be closed. We expect to extend the IxFree library and overcome this limitation in the Coq formalization.

6 PROVING EXAMPLE EQUIVALENCES

We demonstrate that the logical-relations model allows us to prove refinement and equivalence results that would not hold if algebraic effects were not tunneled. Beyond the usefulness of equivalence for programmer reasoning, such equivalence results could be used to justify the soundness of compiler transformations on effectful programs.

Example 1. In this example, we show that clients of an effect-polymorphic abstraction cannot cause implementation details of the abstraction to leak out. We assume that $\lambda_{⇩⇧}$ has a second base type $\mathbb{N}$ with the operator $+$.

Let f be a variable with an effect-polymorphic type $T \stackrel{def}{=} \forall\alpha.\, (\mathbb{N} \to [\mathbb{N}]_{\alpha}) \to [\mathbb{N}]_{\alpha}$. Our goal is to prove the following two terms contextually equivalent:

$$\begin{aligned} t_1 &\stackrel{def}{=} \mathsf{f}\ [\varnothing]\ (\lambda \mathsf{x}{:}\mathbb{N}.\ \mathsf{x} + \mathsf{x}) \\ t_2 &\stackrel{def}{=} \textbf{let}\ \mathsf{g} : \Pi_{\mathsf{h}:\mathbb{F}}\, \mathbb{N} \to [\mathbb{N}]_{\mathsf{h}.\textbf{lbl}} = \lambda \mathsf{h}{:}\mathbb{F}.\ \lambda \mathsf{x}{:}\mathbb{N}.\ ⇧\, \mathsf{h}\ \mathsf{x}\ \textbf{in} \\ &\phantom{\stackrel{def}{=}}\ ⇩^{\ell}_{[\mathbb{N}]_\varnothing}\ (\lambda \mathsf{h}{:}\mathbb{F}.\ \mathsf{f}\ [\mathsf{h}.\textbf{lbl}]\ (\mathsf{g}\ \mathsf{h}))\ H^{\ell} \end{aligned}$$

where $H \stackrel{def}{=} \textbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ \mathsf{k}\ (\mathsf{x} + \mathsf{x})$ and $op(\mathbb{F}) = \mathbb{N} \to \mathbb{N}$. The second term t_2 corresponds to the following program written using the `try-with` construct, assuming the effect operation is named `twice`:

```
effect 𝔽 { twice(ℕ) : ℕ }
val g = fun(x : ℕ) : ℕ / 𝔽 { return twice(x) }
try { f(g) } with twice(x) { resume (x+x) }
```

Notice that this equivalence should apply to all possible (well-typed) implementations of f, so even if the implementation handles $\mathbb{F}$ internally, the clients are unable to make different observations. As a result, equivalence results of this kind ensure the correctness of compiler transformations that optimize away uses of effects like that in t_2.

By the SOUNDNESS theorem, it suffices to show that t_1 and t_2 are logically equivalent. Below we show the logical refinement $\varnothing \mid \varnothing \mid \mathsf{f}{:}T \mid \varnothing \vdash t_1 \preccurlyeq_{log} t_2 : [\mathbb{N}]_\varnothing$ holds; the proof of the other direction is similar. By the definition of logical refinement ($\preccurlyeq_{log}$), we need to show for any f_1 and f_2 in the logical relation $\mathcal{V}[\![T]\!]^{\varnothing}_{\varnothing}$, the terms $t_1\,\{f_1/\mathsf{f}\}$ and $t_2\,\{f_2/\mathsf{f}\}$ are in the logical relation $\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]^{\varnothing}_{\varnothing}$. Notice that we can make reduction steps on $t_2\,\{f_2/\mathsf{f}\}$. So applying Lemma 3, our goal becomes

$$\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]^{\varnothing}_{\varnothing}\Big(f_1\ [\varnothing]\ (\lambda \mathsf{x}{:}\mathbb{N}.\ \mathsf{x} + \mathsf{x}),\ ⇩^{\ell}_{[\mathbb{N}]_\varnothing}\ f_2\ [\ell]\ \big(\lambda \mathsf{x}{:}\mathbb{N}.\ ⇧ H^{\ell}\ \mathsf{x}\big)\Big) \tag{1}$$

We can show a result slightly different from (1): we will show that the terms in (1) are related by $\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]^{\varnothing}_{\delta}$ instead, where δ contains the mapping $\alpha \mapsto \langle \varnothing, \ell, \phi \rangle$ and ϕ is the interpretation specifically chosen for α in this example:

$$\phi = \Big\{ \Big\langle (\lambda \mathsf{x}{:}\mathbb{N}.\ \mathsf{x} + \mathsf{x})\ n,\ \big(\lambda \mathsf{x}{:}\mathbb{N}.\ ⇧ H^{\ell}\ \mathsf{x}\big)\ n,\ \{\langle 2n, 2n \rangle\},\ \varnothing,\ \ell \Big\rangle\ \Big|\ n \in \mathbb{N} \Big\}$$

Having this result, we can use a weakening lemma (omitted) to obtain (1). Here, the presence of effect polymorphism allows us to interpret α in arbitrary ways, but as we shall see, this particular choice of ϕ allows us to establish logical relatedness. To obtain this result, we apply Lemma 1 with evaluation contexts $[\cdot]$ and $⇩^{\ell}_{[\mathbb{N}]_\varnothing}\ [\cdot]$:

- We want to show $\mathcal{K}_{\mathcal{T}}[\![[\mathbb{N}]_\alpha \rightsquigarrow [\mathbb{N}]_\varnothing]\!]_\delta^\varnothing \left([\cdot],\, \Downarrow^\ell [\cdot]\right)$. We apply the [Löb] rule from Section 5.1: to prove this goal, we are allowed to assume

$$\triangleright\, \mathcal{K}_{\mathcal{T}}[\![[\mathbb{N}]_\alpha \rightsquigarrow [\mathbb{N}]_\varnothing]\!]_\delta^\varnothing \left([\cdot],\, \Downarrow^\ell [\cdot]\right) \tag{2}$$

  Unfolding the definition of $\mathcal{K}_{\mathcal{T}}$ generates the following two goals:

  (a) We want to show for any v_1 and v_2 in the relation $\mathcal{V}[\![\mathbb{N}]\!]_\delta^\varnothing$, the terms v_1 and $\Downarrow^\ell v_2$ are related by $\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]_\delta^\varnothing$. This is immediate, because the right-hand side evaluates to v_2 and the value relation is included in the term relation (Lemma 4).

  (b) We want to show for any $K_1[s_1]$ and $K_2[s_2]$ in the relation $\mathcal{S}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing$, the terms $K_1[s_1]$ and $\Downarrow^\ell K_2[s_2]$ are related by $\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]_\delta^\varnothing$. Unfolding the definition of $\mathcal{S}$, we know there exists an outcome relation ψ such that

    (i) $\mathcal{U}[\![\alpha]\!]_\delta^\varnothing \left(s_1,\, s_2,\, \psi,\, \overline{\ell_1},\, \overline{\ell_2}\right)$,

    (ii) $\forall i.\ \ell_1^{(i)} \curvearrowright K_1$ and $\forall i.\ \ell_2^{(i)} \curvearrowright K_2$, and

    (iii) $\forall s_1', s_2'.\ \psi\left(s_1',\, s_2'\right) \Rightarrow \triangleright\, \mathcal{T}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing \left(K_1\left[s_1'\right],\, K_2\left[s_2'\right]\right)$.

  Since we interpret α as ϕ (i.e., $\mathcal{U}[\![\alpha]\!]_\delta^\varnothing \equiv \phi$), we know s_1, s_2, ψ, $\overline{\ell_1}$, and $\overline{\ell_2}$ are precisely the terms, relation, and labels in ϕ. Thus we need to show

$$\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]_\delta^\varnothing \left(K_1[(\lambda \mathsf{x}\!:\!\mathbb{N}.\, \mathsf{x}+\mathsf{x})\ n],\, \Downarrow^\ell K_2\left[\left(\lambda \mathsf{x}\!:\!\mathbb{N}.\, \Uparrow H^\ell\ \mathsf{x}\right)\ n\right]\right)$$

  Making evaluation steps on both sides, the goal becomes $\triangleright\, \mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]_\delta^\varnothing \left(K_1[2n],\, \Downarrow^\ell K_2[2n]\right)$. The new goal is guarded by the $\triangleright$ modality because evaluation occurred in the first computation. The new proof context is as follows, where the first assumption is the Löb induction hypothesis (2):

$$\frac{\triangleright\, \mathcal{K}_{\mathcal{T}}[\![[\mathbb{N}]_\alpha \rightsquigarrow [\mathbb{N}]_\varnothing]\!]_\delta^\varnothing \left([\cdot],\, \Downarrow^\ell [\cdot]\right) \qquad \forall s_1', s_2'.\ \psi\,(s_1',\, s_2') \Rightarrow \triangleright\, \mathcal{T}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing\ (K_1[s_1'],\, K_2[s_2'])}{\triangleright\, \mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]_\delta^\varnothing \left(K_1[2n],\, \Downarrow^\ell K_2[2n]\right)}$$

  We already have $\psi\,(2n,\, 2n)$, so $\triangleright\, \mathcal{T}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing\ (K_1[2n],\, K_2[2n])$ holds. Now we can apply rule [Mono] from Section 5.1: the presence of the $\triangleright$ modality in the goal cancels out the occurrences of $\triangleright$ in the assumptions. The new goal then follows from the definition of $\mathcal{K}_{\mathcal{T}}$.

- We are left to show $\mathcal{T}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing \left(f_1\ [\varnothing]\ (\lambda \mathsf{x}\!:\!\mathbb{N}.\, \mathsf{x}+\mathsf{x}),\, f_2\ [\ell]\ \left(\lambda \mathsf{x}\!:\!\mathbb{N}.\, \Uparrow H^\ell\ \mathsf{x}\right)\right)$. By the hypothesis $\mathcal{V}[\![\forall\alpha.\, (\mathbb{N} \to [\mathbb{N}]_\alpha) \to [\mathbb{N}]_\alpha]\!]_\varnothing^\varnothing\ (f_1,\, f_2)$ and by the definition of $\mathcal{V}$, we have that the terms $f_1\ [\varnothing]$ and $f_2\ [\ell]$ are in the relation $\mathcal{T}\left[\!\left[[(\mathbb{N} \to [\mathbb{N}]_\alpha) \to [\mathbb{N}]_\alpha]_\varnothing \right]\!\right]_\delta^\varnothing$. Because the logical relation is compatible with the typing rule for applications, it suffices to show that the values that $f_1\ [\varnothing]$ and $f_2\ [\ell]$ are applied to (i.e., $\lambda \mathsf{x}:\mathbb{N}.\, \mathsf{x}+\mathsf{x}$ and $\lambda \mathsf{x}:\mathbb{N}.\, \Uparrow H^\ell\ \mathsf{x}$) are in the relation $\mathcal{V}[\![(\mathbb{N} \to [\mathbb{N}]_\alpha)]\!]_\delta^\varnothing$, which by definition means applications of these two abstractions to the same natural number are in the term relation $\mathcal{T}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing$. By Lemma 4, we show the applications are actually in the smaller $\mathcal{S}$ relation:

$$\mathcal{S}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing \left((\lambda \mathsf{x}\!:\!\mathbb{N}.\, \mathsf{x}+\mathsf{x})\ n,\, \left(\lambda \mathsf{x}\!:\!\mathbb{N}.\, \Uparrow H^\ell\ \mathsf{x}\right)\ n\right)$$

  With the evaluation contexts being $[\cdot]$, the following conditions are straightforward to show:

  (i) $\mathcal{U}[\![\alpha]\!]_\delta^\varnothing \left((\lambda \mathsf{x}\!:\!\mathbb{N}.\, \mathsf{x}+\mathsf{x})\ n,\, \left(\lambda \mathsf{x}\!:\!\mathbb{N}.\, \Uparrow H^\ell\ \mathsf{x}\right)\ n,\, \{\langle 2n,\, 2n\rangle\},\, \varnothing,\, \ell\right)$,

  (ii) $\ell \curvearrowright [\cdot]$, and

  (iii) for any s_1' and s_2' related by $\{\langle 2n, 2n\rangle\}$, $\triangleright\, \mathcal{T}[\![[\mathbb{N}]_\alpha]\!]_\delta^\varnothing \left(s_1',\, s_2'\right)$.

Example 2. In this example, we show tunneled algebraic effects preserve the abstraction of handler polymorphism.

Let f be a variable with a handler-polymorphic type $\Pi_{\mathsf{h}:\mathbb{F}}\ (\mathbb{N} \to [\mathbb{N}]_{\mathsf{h}.\mathbf{lbl}}) \to [\mathbb{N}]_{\mathsf{h}.\mathbf{lbl}}$. Our goal is to prove the following two terms contextually equivalent:

$$t_1 \overset{def}{=} \Downarrow^\ell_{[\mathbb{N}]_\varnothing} (\lambda \mathsf{h}\!:\!\mathbb{F}.\, \mathsf{f}\ \mathsf{h}\ (\lambda \mathsf{x}\!:\!\mathbb{N}.\, \mathsf{x}+\mathsf{x}))\ H^\ell$$

$$t_2 \overset{def}{=} \Downarrow^\ell_{[\mathbb{N}]_\varnothing} (\lambda \mathsf{h}\!:\!\mathbb{F}.\, \mathsf{f}\ \mathsf{h}\ (\lambda \mathsf{x}\!:\!\mathbb{N}.\, \Uparrow \mathsf{h}\ \mathsf{x}))\ H^\ell$$

where $H \stackrel{def}{=} \textbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ \mathsf{k}\ (\mathsf{x}+\mathsf{x})$ and $op(\mathbb{F}) = \mathbb{N} \to \mathbb{N}$. Again, this equivalence is expected to hold regardless of the implementation of f, which is free to handle $\mathbb{F}$ internally.

The proof is structured in an analogous way to that in Example 1: we apply Lemma 1 and prove that the evaluation contexts $\Downarrow^{\ell}[\cdot]$ and $\Downarrow^{\ell}[\cdot]$ are in the relation $\mathcal{K}_{\mathcal{T}}[\![[\mathbb{N}]_{\mathsf{h}.\mathbf{lbl}} \rightsquigarrow [\mathbb{N}]_{\varnothing}]\!]^{\rho}_{\varnothing}$ and that the application terms $f_1\ H^{\ell}\ (\lambda\mathsf{x}:\mathbb{N}.\ \mathsf{x}+\mathsf{x})$ and $f_2\ H^{\ell}\ \left(\lambda\mathsf{x}:\mathbb{N}.\Uparrow H^{\ell}\ \mathsf{x}\right)$, where f_1 and f_2 are (related) substitutions for f, are in the relation $\mathcal{T}[\![[\mathbb{N}]_{\mathsf{h}.\mathbf{lbl}}]\!]^{\rho}_{\varnothing}$. Here $\rho \stackrel{def}{=} \mathsf{h} \mapsto \left\langle H^{\ell}, H^{\ell}, \mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\varnothing}\right\rangle$, and $\mathcal{H}[\![\mathbb{F}]\!]\ \left(H^{\ell},\ H^{\ell},\ \mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\varnothing}\right)$ holds by Parametricity. The new element in this proof is the interpretation of the effect $\mathsf{h}.\mathbf{lbl}$. In particular, showing the subgoal

$$\mathcal{S}[\![[\mathbb{N}]_{\mathsf{h}.\mathbf{lbl}}]\!]^{\rho}_{\varnothing}\ \left((\lambda\mathsf{x}:\mathbb{N}.\ \mathsf{x}+\mathsf{x})\ n,\ \left(\lambda\mathsf{x}:\mathbb{N}.\Uparrow H^{\ell}\ \mathsf{x}\right)\ n\right)$$

involves showing $\mathcal{U}[\![\mathsf{h}.\mathbf{lbl}]\!]^{\rho}_{\varnothing}\ \left((\lambda\mathsf{x}:\mathbb{N}.\ \mathsf{x}+\mathsf{x})\ n,\ \left(\lambda\mathsf{x}:\mathbb{N}.\Uparrow H^{\ell}\ \mathsf{x}\right)\ n,\ \{\langle 2n, 2n\rangle\},\ \ell,\ \ell\right)$, which can be verified as follows:

$$\begin{aligned}
&\forall K.\ \ell \curvearrowright K \Rightarrow\ \Downarrow^{\ell} K[(\lambda\mathsf{x}:\mathbb{N}.\ \mathsf{x}+\mathsf{x})\ n] \longrightarrow^{+} \Downarrow^{\ell} K[2n]\\
&\forall K.\ \ell \curvearrowright K \Rightarrow\ \Downarrow^{\ell} K\left[\left(\lambda\mathsf{x}:\mathbb{N}.\Uparrow H^{\ell}\ \mathsf{x}\right)\ n\right] \longrightarrow^{*} \Downarrow^{\ell} K[2n]
\end{aligned}$$

Note that the corresponding definition in Figure 14 requires the first computation to take at least one reduction step, so when verifying that the evaluation contexts are in the $\mathcal{K}_{\mathcal{T}}$ relation, the [mono] rule allows shifting reasoning to a future world where the Löb induction hypothesis applies.

7 RELATED WORK

Previous work proposes ways to make algebraic effects composable. Leijen [27] suggests using an `inject` function to prevent client code from meddling with the effect-handling internals of library functions. Applying `inject` to a computation causes effects raised from that computation to bypass the innermost handler enclosing it. Biernacki et al. [7] propose a "lift" operator that works in a similar fashion: computations surrounded by a lift operator $[\cdot]_{\mathbb{F}}$ bypass the innermost effect handler for $\mathbb{F}$. The programmer can use `inject` or lift to prevent effects of a client-provided function from being intercepted by the effect-polymorphic, higher-order function that applies it. Both of these type systems use effect rows and row polymorphism, and distinguish different occurrences of the same effect name in a row.

The very use of effect rows in these approaches does not seem to be without limitations. In particular, it poses challenges to composing polymorphic effects. For example, because α, β is not a legal effect row, this effect-polymorphic higher-order function type does not seem to be expressible using effect rows: $\forall\alpha.\forall\beta.\ ((T_1 \to [T_2]_{\alpha}) \to [T_3]_{\beta}) \to (T_1 \to [T_2]_{\alpha}) \to [T_3]_{\alpha,\beta}$.

Biernacki et al. [7] show that effect polymorphism in a core language equipped with the lift operator satisfies parametricity; we borrow useful techniques from their logical-relations definition. The type system of Biernacki et al. poses restrictions on "subeffecting" (cf. subtyping): it rejects—by fiat—an effect variable α as a subeffect of $\mathbb{F}, \alpha$. The absence of accidental handling hinges upon this restriction: the programmer must thread lift operators through effect-polymorphic code to please the type checker. For example, function `fiterate` from Section 2 would not type-check in their system because the effect of `f(x)` (i.e., effect variable `E`) is not a subeffect of `Yield[X], E`. The programmer would have to choose between (a) declaring variable `f` with type `X` $\to$ `bool /Yield[X], E`, and (b) surrounding `f(x)` with a lift operator. In contrast, because it rests on the intuitive principle that code should only handle effects it is locally aware of, tunneling requires no essential changes to effect-polymorphic code.

Zhang et al. [47] propose an alternate semantics for exceptions in their Genus language, in which exceptions are tunneled through contexts that are not statically aware of them. While we build on this insight, this prior work is limited to exceptions rather than more general algebraic effects, and importantly, the mechanism is not shown formally to be abstraction-safe. The kind of exception polymorphism it supports is also more limited: functions are polymorphic in the latent exceptions of only those types that are annotated `weak`. It is argued that trading `weak` annotations for explicit effect variables reduces annotation burden. However, this approach makes it cumbersome, if not impossible, to define exception-polymorphic *data structures*, such as the `cachingFun` class in Section 3.4. The `weak` annotations are essentially a mechanism for region-capability effects: values of weak types have a stack discipline and thus can only be used in a second-class way, but data structures require a finer-grained notion of region capability.

Functional programming languages like ML and Haskell do not statically check that exceptions are handled, so we do not consider them fully type-safe. Interestingly, accidental handling can be avoided in SML, because SML exception types are generative [32] and because a handler can only handle lexically visible exception types. However, the type system does not ensure that accidental handling is avoided or that exceptions are handled at all. Bračevac et al. [10] observe the need to disambiguate handlers for invocations of the same algebraic effect operation. Compared with their proposed solution of generative effect signatures, tunneling addresses the issue straightforwardly: handlers can be specified explicitly for each invocation of the effect operation.

Brachthäuser and Schuster [8] encode algebraic effect handlers as a Scala library named Effekt. Like our use of handler polymorphism, the encoding passes handlers down to the place where effect operations are invoked, using Scala's *implicits* feature [35] and in particular, implicit function types [34], to resolve implicit arguments as handler objects in scope. Clients of Effekt do not have to worry about accidental handling, but this approach does not guarantee the absence of run-time errors. In addition to the handling code, a stack-marking *prompt* must be passed down too, so that when the effect operation is invoked, the continuation up to the prompt is captured and passed to the handling code. But there is no static checking that the prompt obeys the stack discipline—type-safety relies on client code using the library in a disciplined way.

It is hypothesized that this safety issue could be remedied by using the `@local` annotation provided in a Scala extension [36]. Parameters of functions and local variables can be annotated `@local`, making them second-class. In contrast to the Genus `weak` annotation [47], `@local` is applied to uses of types (instead of definitions of types), so it seems no lighter-weight than explicit effect variables. Like the `weak` annotations, `@local` cannot offer the fine-grained notion of region capability needed to express effect-polymorphic data structures.

Our use of capability effects to ensure soundness is adapted from work on region-based memory management [12, 21, 44]. A capability is a set of live memory regions. To prevent accesses to deallocated memory regions, computations are typed with capability effects that specify the set of regions they might access. We apply this idea to ensure continuations of handling code are accessible. Our type system is simpler than a full-fledged region type system because safety concerns only lexical regions delimited by effect handlers.

The problem of accidentally handled effects generalizes the problem of variable capture in early programming languages (e.g., Lisp) that supported dynamically scoped variables. Dynamically scoped variables do not have to be dynamically typed; Lewis et al. [29] provide a type system for them, treating them as implicit parameters. To avoid variable capture, Lewis et al. ban the use of implicitly parameterized functions as first-class values, losing the extensibility that makes dynamically scoped variables attractive. Tunneled algebraic effects offer abstraction-safe dynamically scoped variables without sacrificing their expressive power.

Kammar et al. [25] distinguish between deep and shallow semantics for handlers. A shallow handler is discarded after it is first invoked, while a deep handler can continue to handle the rest of the computation it envelops. Handlers for tunneled algebraic effects are deep. Shallow handlers pose challenges to modular reasoning, because it is difficult to reason statically about how effects raised from the rest of the computation are handled.

The effect constructs in our core language are essentially a pair of delimited control operators [13, 19]. With delimited control, one operator $\mathcal{C}$ (cf. ⇧ in $\lambda_{⇩⇧}$) captures the continuation delimited by a corresponding operator of the other kind $\mathcal{D}$ (cf. ⇩ in $\lambda_{⇩⇧}$). Among the variety of previous delimited control operators, ours are closest to those with named prompts [17, 22]. Rather than pairing a $\mathcal{C}$ operation with the *dynamically* closest enclosing $\mathcal{D}$, these mechanisms allow uses of $\mathcal{D}$ to be named and consequently referenced by invocations of $\mathcal{C}$, enabling static reasoning. Although embedded in statically typed languages, the earlier mechanisms do not guarantee type safety—a C operation can go unhandled.

8 CONCLUSION

We have argued that tunneling is the right semantics for algebraic effects because, as we have shown formally, it makes them abstraction-safe, preserving modular reasoning. Because algebraic effects generalize other mechanisms such as exceptions, dynamically scoped variables, and coroutines, the tunneling semantics fixes not only algebraic effects generically, but also the design of several specific language features. We have provided a strong foundation for the design of algebraic-effect mechanisms that are not only type-safe, but also abstraction-safe. Our new semantics should be a useful guide for future language designs and also motivate support for algebraic effects in mainstream languages.

ACKNOWLEDGMENTS

We thank Josh Acay, Sam Lindley, Craig McLaughlin, and Drew Zagieboylo for their helpful feedback on this paper, along with our shepherd and the anonymous reviewers. We also thank Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski for making their software available. This research was supported by NSF grant 1513797 and NASA grant NNX16AB09G but does not necessarily represent the opinions of these funding agencies.

REFERENCES

[1] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *15th European Symposium on Programming*, 2006. Extended/corrected version available as Harvard University TR-01-06.

[2] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5), September 2001.

[3] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *34th ACM Symp. on Principles of Programming Languages (POPL)*, 2007.

[4] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4, December 2014.

[5] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1), 2015.

[6] Nick Benton and Uri Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 1–12, 2007.

[7] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM on Programming Languages*, 2(POPL), December 2017.

[8] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: Extensible algebraic effects in Scala (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, 2017.

[9] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Algebraic effects for the masses. *Proc. ACM on Programming Languages*, 2(OOPSLA), October 2018.

[10] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile event correlation with algebraic effects. *Proc. ACM on Programming Languages*, 2(ICFP), August 2018.

[11] Coq 8.7. The Coq proof assistant. http://coq.inria.fr. Version 8.7.

[12] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, 1999.

[13] Olivier Danvy and Andrzej Filinski. Abstracting control. In *ACM Conf. on LISP and Functional Programming*, pages 151–160, 1990.

[14] Derek Dreyer. Milner award lecture: The type soundness theorem that you really want to prove (and now you can). In *45th ACM Symp. on Principles of Programming Languages (POPL)*, 2018.

[15] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *24th Annual IEEE Symposium on Logic In Computer Science (LICS)*, 2009.

[16] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

[17] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, November 2007. ISSN 0956-7968.

[18] Matthias Felleisen. *The calculi of λ-v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages.* PhD thesis, Indiana University, Indianapolis, IN, USA, 1987.

[19] Mattias Felleisen. The theory and practice of first-class prompts. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 180–190, 1988.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, MA, 1994. ISBN 0-201-63361-2.

[21] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 282–293. ACM Press, 2002.

[22] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In *7th Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.

[23] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, 2016.

[24] Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *25th Annual IEEE Symposium on Logic In Computer Science (LICS)*, 2010.

[25] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *18th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2013.

[26] Donald Ervin Knuth. *The TEXbook.* Addison-Wesley Reading, 1984.

[27] Daan Leijen. Koka: Programming with row polymorphic effect types. In *5th Workshop on Mathematically Structured Functional Programming*. EPTCS, 2014.

[28] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017.

[29] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In *27th ACM Symp. on Principles of Programming Languages (POPL)*, 2000.

[30] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017.

[31] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[32] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990. ISBN 978-0262631327.

[33] J. H. Morris, Jr. *Lambda-Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1968.

[34] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and applications of implicit function types. *Proc. ACM on Programming Languages*, 2(POPL), December 2017.

[35] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2010.

[36] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. Gentrification gone too far? Affordable 2nd-class values for fun and (co-)effect. In *2016 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

[37] Andrew M Pitts and Ian Stark. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, pages 227–273, 1998.

[38] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, Feb 2003.

[39] Gordon Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, December 2013.

[40] Piotr Polesiuk. IxFree: Step-indexed logical relations in Coq. In *3rd International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[41] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[42] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *26th European Conf. on Object-Oriented Programming*, 2012.

[43] Guy L. Steele, Jr. *Common LISP: the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.

[44] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[45] Philip Wadler. Theorems for free! In *4th Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.

[46] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115 (1):38–94, 1994. ISSN 0890-5401.

[47] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. Accepting blame for safe tunneled exceptions. In *37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 281–295, June 2016. URL http://www.cs.cornell.edu/andru/papers/exceptions.

A STATIC SEMANTICS OF $\lambda_{⇩⇧}$

A.1 Term and Handler Well-Formedness

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}}$$

[T-UNIT] $$\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash () : [\mathbb{1}]_{\varnothing}$$

[T-VAR] $$\frac{\Gamma(\mathsf{x}) = T}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{x} : [T]_{\varnothing}}$$

[T-ABS] $$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}S \mid \Xi \vdash t : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{x}{:}S.\, t : [S \to [T]_{\overline{e}}]_{\varnothing}}$$

[T-APP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [S \to [T]_{\overline{e}}]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t\ s : [T]_{\overline{e}}}$$

[T-LET] $$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s : [S]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}S \mid \Xi \vdash t : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathbf{let}\ \mathsf{x}{:}S = s\ \mathbf{in}\ t : [T]_{\overline{e}}}$$

[T-EABS] $$\frac{\Delta,\ \alpha \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\varnothing}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Lambda\alpha.\, t : [\forall\alpha.\, T]_{\varnothing}}$$

[T-EAPP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [\forall\alpha.\, T]_{\overline{e_1}} \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e_2^{(i)}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t\ [\overline{e_2}] : [T\,\{\overline{e_2}/\alpha\}]_{\overline{e_1}}}$$

[T-HABS] $$\frac{\Delta,\ \mathsf{h}{:}\mathbb{F} \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{h}{:}\mathbb{F}.\, t : [\Pi_{\mathsf{h}:\mathbb{F}}\ [T]_{\overline{e}}]_{\varnothing}}$$

[T-HAPP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : \left[\Pi_{\mathsf{h}:\mathbb{F}}\ [T]_{\overline{e_1}}\right]_{\overline{e_2}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e_3}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t\ h : [T\,\{h/\mathsf{h}\}]_{\overline{e_1}\{h/\mathsf{h}\},\ \overline{e_2}}}$$

[T-UP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e \qquad op(\mathbb{F}) = T \to S}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash ⇧h : [T \to [S]_{e}]_{\varnothing}}$$

[T-DOWN] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi,\ \ell{:}[T]_{\overline{e}} \vdash t : [T]_{\overline{e},\ell} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash ⇩^{\ell}_{[T]_{\overline{e}}}\ t : [T]_{\overline{e}}}$$

[T-SUB] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T_1]_{\overline{e_1}} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \le T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \le \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T_2]_{\overline{e_2}}}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e}$$

[T-HVAR] $$\frac{\mathrm{P}(\mathsf{h}) = \mathbb{F}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{h} : \mathbb{F} \mid \mathsf{h}.\mathbf{lbl}}$$

[T-HDEF] $$\frac{\Xi(\ell) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2 \\ \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}T_1,\ \mathsf{k}{:}T_2 \to [S]_{\overline{e}} \mid \Xi \vdash t : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\, t\right)^{\ell} : \mathbb{F} \mid \ell}$$

A.2 Type and Effect Well-Formedness

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash T}$$

$$\Delta \mid \mathrm{P} \mid \Xi \vdash \mathbb{1} \qquad \frac{\Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e^{(i)}}{\Delta \mid \mathrm{P} \mid \Xi \vdash T \to [S]_{\overline{e}}}$$

$$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Xi \vdash T}{\Delta \mid \mathrm{P} \mid \Xi \vdash \forall \alpha.\, T} \qquad \frac{\Delta \mid \mathrm{P}, \mathsf{h}{:}\mathbb{F} \mid \Xi \vdash T \qquad (\forall i)\ \Delta \mid \mathrm{P}, \mathsf{h}{:}\mathbb{F} \mid \Xi \vdash e^{(i)}}{\Delta \mid \mathrm{P} \mid \Xi \vdash \Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash e}$$

$$\frac{\alpha \in \Delta}{\Delta \mid \mathrm{P} \mid \Xi \vdash \alpha} \qquad \frac{\ell \in \mathit{domain}(\Xi)}{\Delta \mid \mathrm{P} \mid \Xi \vdash \ell} \qquad \frac{\mathsf{h} \in \mathit{domain}(\mathrm{P})}{\Delta \mid \mathrm{P} \mid \Xi \vdash \mathsf{h}.\mathbf{lbl}}$$

A.3 Partial Orders on Types and Effect Sequences

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash T \leq S}$$

$$\Delta \mid \mathrm{P} \mid \Xi \vdash \mathbb{1} \leq \mathbb{1} \qquad \frac{\Delta \mid \mathrm{P} \mid \Xi \vdash T_2 \leq T_1 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S_1 \leq S_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \to [S_1]_{\overline{e_1}} \leq T_2 \to [S_2]_{\overline{e_2}}}$$

$$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2}{\Delta \mid \mathrm{P} \mid \Xi \vdash \forall \alpha.\, T_1 \leq \forall \alpha.\, T_2} \qquad \frac{\Delta \mid \mathrm{P}, \mathsf{h}{:}\mathbb{F} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P}, \mathsf{h}{:}\mathbb{F} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Xi \vdash \Pi_{\mathsf{h}:\mathbb{F}}\, [T_1]_{\overline{e_1}} \leq \Pi_{\mathsf{h}:\mathbb{F}}\, [T_2]_{\overline{e_2}}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_2 \leq T_3}{\Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_3}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}$$

$$\frac{(\forall j, \exists i)\ e_1^{(j)} = e_2^{(i)} \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e_2^{(i)}}{\Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}$$

A.4 Well-formedness of Program Contexts

$$\boxed{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T}$$

$$\vdash [\cdot] : \varnothing \mid \varnothing \mid \varnothing \mid \varnothing \mid [T]_{\varnothing} \rightsquigarrow T \qquad \frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T \to [S]_{\overline{e}}]_{\varnothing} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T}{\vdash C[\lambda \mathsf{x}{:}T.\,[\cdot]] : \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}T \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\begin{array}{c}\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T' \\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}\end{array}}{\vdash C[[\cdot]\ t] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T \to [S]_{\overline{e}}]_{\overline{e}} \rightsquigarrow T'} \qquad \frac{\begin{array}{c}\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T' \\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T \to [S]_{\overline{e}}]_{\overline{e}}\end{array}}{\vdash C[t\ [\cdot]] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}S \mid \Xi \vdash t : [T]_{\overline{e}}}{\vdash C[\mathbf{let}\ \mathsf{x}{:}S = [\cdot]\ \mathbf{in}\ t] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s : [S]_{\overline{e}}}{\vdash C[\mathbf{let}\ \mathsf{x}{:}S = s\ \mathbf{in}\ [\cdot]] : \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}S \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [\forall \alpha.\, T]_{\varnothing} \rightsquigarrow T'}{\vdash C[\Lambda \alpha.\,[\cdot]] : \Delta,\ \alpha \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\varnothing} \rightsquigarrow T'} \qquad \frac{\begin{array}{c}\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T\{\overline{e_2}/\alpha\}]_{\overline{e_1}} \rightsquigarrow T' \\ (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e_2^{(i)}\end{array}}{\vdash C[[\cdot]\ \lfloor \overline{e_2} \rfloor] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [\forall \alpha.\, T]_{\overline{e_1}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [\Pi_{\mathsf{h}:\mathbb{F}}\,[T]_{\overline{e}}]_{\varnothing} \rightsquigarrow T'}{\vdash C[\lambda \mathsf{h}{:}\mathbb{F}.\,[\cdot]] : \Delta \mid \mathrm{P},\ \mathsf{h}{:}\mathbb{F} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T'} \qquad \frac{\begin{array}{c}\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T\{h/\mathsf{h}\}]_{\overline{e_1}\{h/\mathsf{h}\},\ \overline{e_2}} \rightsquigarrow T' \\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid e\end{array}}{\vdash C[[\cdot]\ h] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid \left[\Pi_{\mathsf{h}:\mathbb{F}}\,[T]_{\overline{e_1}}\right]_{\overline{e_2}} \rightsquigarrow T'}$$

$$\frac{\begin{array}{c}\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T\{\ell/\mathsf{h}.\mathbf{lbl}\}]_{\overline{e_1}\{\ell/\mathsf{h}.\mathbf{lbl}\},\overline{e_2}} \rightsquigarrow T' \\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : \left[\Pi_{\mathsf{h}:\mathbb{F}}\,[T]_{\overline{e_1}}\right]_{\overline{e_2}} \qquad \Xi(\ell) = [S]_{\overline{e_3}} \qquad op(\mathbb{F}) = T_1 \to T_2\end{array}}{\vdash C\left[t\ \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\,[\cdot]\right)^{\ell}\right] : \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}T_1,\ \mathsf{k}{:}T_2 \to [S]_{\overline{e_3}} \mid \Xi \mid [S]_{\overline{e_3}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T_1 \to [T_2]_{\ell}]_{\varnothing} \rightsquigarrow T' \qquad \Xi(\ell) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2}{\vdash C\left[\Uparrow \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\,[\cdot]\right)^{\ell}\right] : \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}T_1,\ \mathsf{k}{:}T_2 \to [S]_{\overline{e}} \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\vdash C\left[\Downarrow^{\ell}_{[T]_{\overline{e}}}\,[\cdot]\right] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi,\ \ell \mid [T]_{\overline{e},\ell} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T_2]_{\overline{e_2}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T_1]_{\overline{e_1}} \rightsquigarrow T'}$$

B COMPATIBILITY LEMMAS

Lemma 7 (Compatibility with [t-sub]).

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T_1]_{\overline{e_1}} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T_2]_{\overline{e_2}}}$$

Lemma 8 (Compatibility with [t-unit]).

$$\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash () \preccurlyeq_{log} () : [\mathbb{1}]_{\varnothing}$$

Lemma 9 (Compatibility with [t-var]).

$$\frac{\Gamma(\mathsf{x}) = T}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{x} \preccurlyeq_{log} \mathsf{x} : [T]_{\varnothing}}$$

Lemma 10 (Compatibility with [t-abs]).

$$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x}:S \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{x}:S.\, t_1 \preccurlyeq_{log} \lambda \mathsf{x}:S.\, t_2 : [S \to [T]_{\overline{e}}]_{\varnothing}}$$

Lemma 11 (Compatibility with [t-app]).

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S \to [T]_{\overline{e}}]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s_1 \preccurlyeq_{log} s_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1\ s_1 \preccurlyeq_{log} t_2\ s_2 : [T]_{\overline{e}}}$$

Lemma 12 (Compatibility with [t-eabs]).

$$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\varnothing}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Lambda\alpha.\, t_1 \preccurlyeq_{log} \Lambda\alpha.\, t_2 : [\forall \alpha.\, T]_{\varnothing}}$$

Lemma 13 (Compatibility with [t-eapp]).

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [\forall \alpha.\, T]_{\overline{e'}} \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e'^{(i)}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1\ [\overline{e}] \preccurlyeq_{log} t_2\ [\overline{e}] : [T\,\{\overline{e}/\alpha\}]_{\overline{e'}}}$$

Lemma 14 (Compatibility with [t-habs]).

$$\frac{\Delta \mid \mathrm{P}, \mathsf{h}:\mathbb{F} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{h}:\mathbb{F}.\, t_1 \preccurlyeq_{log} \lambda \mathsf{h}:\mathbb{F}.\, t_2 : [\Pi_{\mathsf{h}:\mathbb{F}}\ [T]_{\overline{e}}]_{\varnothing}}$$

Lemma 15 (Compatibility with [t-happ]).

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : \left[\Pi_{\mathsf{h}:\mathbb{F}}\ [T]_{\overline{e'}}\right]_{\overline{e''}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid e}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1\ [h_1] \preccurlyeq_{log} t_2\ [h_2] : [T\,\{e/\mathsf{h}.\mathbf{lbl}\}]_{\overline{e'}\{e/\mathsf{h}.\mathbf{lbl}\},\, \overline{e''}}}$$

Lemma 16 (Compatibility with [t-up]).

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid e \qquad op(\mathbb{F}) = T \to S}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Uparrow h_1 \preccurlyeq_{log} \Uparrow h_2 : [T \to [S]_e]_\varnothing}$$

Lemma 17 (Compatibility with [t-down]).

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi, \ell : [T]_{\overline{e}} \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}, \ell} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Downarrow^{\ell}_{[T]_{\overline{e}}} t_1 \preccurlyeq_{log} \Downarrow^{\ell}_{[T]_{\overline{e}}} t_2 : [T]_{\overline{e}}}$$

Lemma 18 (Compatibility with [t-hvar]).

$$\frac{\mathrm{P}(\mathsf{h}) = \mathbb{F}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{h} \preccurlyeq_{log} \mathsf{h} : \mathbb{F} \mid \mathsf{h}.\mathbf{lbl}}$$

Lemma 19 (Compatibility with [t-hdef]).

$$\frac{\Xi(\ell) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2 \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x} : T_1, \mathsf{k} : T_2 \to [S]_{\overline{e}} \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_1\right)^{\ell} \preccurlyeq_{log} \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_2\right)^{\ell} : \mathbb{F} \mid \ell}$$