Extending Abstract Effects with Bounds and Algebraic Handlers

Anlun Xu

CMU-CS-20-141

December 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Chair)
Frank Pfenning
Alex Potanin (Victoria University of Wellington)

Submitted in partial fulfillment of the requirements
for the degree of Master of Science.

Keywords: effect systems, algebraic effects and handlers, language-based security, modularity, existential types

This thesis is dedicated to my parents.
For their endless love, support, and encouragement.

Abstract

Effect systems have been a subject of active research for nearly four decades, with the most notable practical example being checked exceptions in programming languages such as Java. The work on effects can be divided into two strands: The *restrictive* approach (e.g., Java's checked exceptions) tracks effects that are already built into the language–such as reading and writing state or exceptions–and provides a way to restrict them. The *denotational* approach, which includes algebraic effects, defines the semantics of computational effects based on primitives. While there are many existing restrictive or denotational effect systems, they are rarely designed with scalability in mind. In this thesis, we design multiple effect systems around the idea of making effect system scalable when developing large and complex software. The first part of our work is a restrictive path-dependent effect system that provides a granular effect hierarchy by allowing abstract effect members to be bounded. This thesis presents a full formalization of the effect system, and provides an implementation as a part of the Wyvern programming language. The second part of our work presents a denotational effect system that supports abstract algebraic effects. This thesis gives a formalization of the system and provides proofs for type soundness and properties of effect abstraction.

Acknowledgments

I would like to thank my advisors, Jonathan Aldrich and Alex Potanin, for their consistent support and guidance during the running of this project. Furthermore, I would like to thank Frank Pfenning, for instructing the course Types and Programming Languages, and providing thorough feedback and critiques. I would also like to thank Darya Melicher for her fundamental contribution to the earlier stage of this work.

Contents

List of Figures

Chapter 1

Introduction

Effect systems have been a subject of active research for nearly four decades, with the most notable practical example being checked exceptions in programming languages such as Java. According to Filinski [7], there are two different views on modeling computational effects in programs: the denotational approach and the restrictive approach.

The denotational approach describes how effectful programs can be translated into a pure program, which can then be evaluated using the standard semantics for pure programs. Works by Moggi [22] show that features from imperative computations, such as exceptions or mutable states, can be mimicked by monads in a pure program. Algebraic effects and handlers [26] are the latest development in this strand of works. Algebraic effects and handlers can express a wide range of computation effects such as nondeterminism, concurrency, state, and input/output [26]. Comparing to the traditional approach that uses general monads, algebraic effects have the advantage of being freely composable. Therefore, algebraic effects have recently been gaining popularity as an approach to model effects in a purely functional setting.

Alternatively, in a restrictive setting of computational effects, effects are considered to be built into the language. Rather than building up new behaviors, the effect systems aim to classify and restrict the use of existing effectful behavior in a language, such as reads and writes to memory, as well as checked exceptions. Restrictive effect systems are widely used for reasoning about security [30], memory effects [17], and concurrency [3, 5, 6].

Abstraction: A requirement for scalable effect systems

Unfortunately, effect systems have not been widely adopted, other than checked exceptions in Java, a feature that is widely viewed as problematic [31]. The root of the problem is that existing effect systems do not provide adequate support for scaling to programs that are larger and have a complex structure. Any adequate solution must support *effect abstraction* and *effect composition*.

Abstraction is key to achieving scale in general, and a principal form of abstraction is abstract types [20]. There are many existing works that achieve information hiding using abstract types, such as SML signatures and abstract type members in Scala [23]. Typically, a module system allows each module to choose what names and entities to export, and what to keep hidden. The exported interface typically does not reveal details of the implementation of a module. By hiding the implementation details, the programmer of the module can be certain that the invariants

within the module cannot be broken by the client. Analogously to type abstraction, we define *effect abstraction* as the ability to define higher-level effects in terms of lower-level effects, and potentially to *hide* that definition from clients of effects.

In large-scale systems, abstraction should be *composable*. For example, a database component might abstract `file.Read` further, exposing it as a higher-level `db.Query` effect to clients. Clients of the database should be oblivious to whether `db.Query` is implemented in terms of a `file.Read` effect or a `network.Access` effect (in the case that the backend is a remote database).

Design of a restrictive effect system in Wyvern

This thesis presents a novel and scalable effect system design that supports bounded effect abstraction, extending the effect system presented by Melicher et al. [19]. The abstraction facility of our effect-system is inspired by type members in languages such as Scala. Just as Scala objects may define type members, in our effect calculus, any object may define one or more *effect members*. An effect member defines a new effect in terms of the lower-level effects that are used to implement it. The set of lower-level effects may be empty in the base case or may include low-level effects that are hard-coded in the system. Type ascription can enable information hiding by concealing the definition of an effect member from the containing object's clients. In addition to completely concealing the definition of an effect, our calculus provides bounded abstraction, which exposes upper or lower bounds of the definition of an effect, while still hiding the definition of it.

Effect polymorphism is a form of parametric polymorphism that allows functions or types to be implemented generically for handling computations with different effects [17]. In systems at a larger scale, there are various possible effects, and each program component may cause different effects. With effect polymorphism, we can write generic code that handles objects with different effects, thereby reducing the amount of replicated code. In practice, we have found that to make effects work well with modules, it is essential to extend effect polymorphism by assigning bounds to effect parameters. We, therefore, introduce *bounded abstract effects*, which allows programmers to define upper and lower bounds both on abstract effects and on polymorphic effect parameters.

Just as Scala's type members can be used to encode parametric polymorphism over types, our effect members and their bounds double as a way to provide bounded effect polymorphism. Instead of explicitly supporting parametric polymorphism using universal types, we follow numerous prior Scala formalisms and use effect members to encode polymorphic methods; this keeps the formal system simpler without giving up expressive power.

Design of a denotational effect system with effect abstraction

This thesis presents a core calculus that supports algebraic effects. The calculus extends the simply typed lambda calculus with algebraic effect operations and handlers and provides the ability to define abstract algebraic effects. Similar to the restrictive effect system, algebraic effect types can be defined in terms of lower-level effects. Effect abstraction in this system ensures that the client of an abstract effect type is not aware of the lower-level effects that implement the abstract effect. Consequently, the client of an abstract effect type would not be able to handle the computation that causes the abstract effect, whose operations are hidden.

Different from the restrictive effect system in Wyvern, which describes the built-in effectful

behavior in the language and does not affect the dynamic semantics of the program, the dynamic semantics of algebraic effects operations depend on the handler that encapsulates it during evaluation. Therefore, the effect system needs to ensure the abstraction does not break during the evaluation of a program. This problem was originally discovered by Biernacki et al. [2], who solved the problem using the technique of coercion. In this paper, we propose the technique of agent-based reasoning, which was originally designed by Grossman et al. [10], as a solution to the problem. The benefit of this approach is that by explicitly dividing modules with hidden information into agents, the system supports syntactic proof for effect-abstraction properties

Outline and Contributions. Chapter 2 introduces the background for both restrictive and denotational effect systems, and discusses the basics of the Wyvern effect system, after which we describe the main contributions of our paper:

- A design of a more expressive effect system for Wyvern. Specifically, ours is the first system to provide the programmer with a general form of bounded effect polymorphism and bounded effect abstraction, supporting upper and lower bounds that are other arbitrary effects. (Section 3.2);
- A precise, formal description of our effect system, and proof of its soundness. Our formal system shows how to generalize and enrich earlier work on path-dependent effects by leveraging the type theory of DOT [1]. (Section 3.3);
- A multi-agent calculus that supports abstraction for algebraic effects, and proof of its type soundness theorems; Our system enables a syntactic proof of the effect-abstraction property. (Section 4.3);
- A multi-agent calculus extended with existential effect types that demonstrates how multi-agent calculus interacts with traditional techniques of type abstraction. (Section 5.3);

The last chapter in the thesis discusses related work and concludes.

Chapter 2

Background

In this chapter, we introduce designs of existing restrictive effect systems and denotational effect systems and discuss their shortcomings.

2.1 A Modular Design of Restrictive Effect Systems

Restrictive Systems were originally proposed by Lucassen [16] to track reads and writes to memory, and then Lucassen and Gifford [17] extended this effect system to support polymorphism. Effects have since been used for a wide variety of purposes, including exceptions in Java [11] and asynchronous event handling [5]. Turbak and Gifford [30] previously proposed effects as a mechanism for reasoning about security, which is the main application that we discuss.

This section describes the effect system of the Wyvern programming language by Melicher et al. [19], which introduces various effect system features such as effect members, effect abstraction, and path-dependent effects. The paper shows that the effect system lays a solid foundation for effect systems that can scale up and can deal with complexities of real-world code.

2.1.1 Running Example

Consider the Wyvern code in Fig. 2.1 that shows a type and a module implementing the logging facility of a text editor application. In the given implementation of the `Logger` type, the `logger` module accesses the log file.[1] All modules of type `Logger` must have two methods: the `readLog` method that returns the content of the log file and the `updateLog` method that appends new entries to the log file. In addition, the `Logger` type declares two *abstract* effects, `ReadLog` and `UpdateLog`, that are produced by the corresponding methods. These effects are abstract because they are not given a definition in the `Logger` type, and so it is up to the module implementing the `Logger` type to define what they mean. The effect names are user-defined, allowing the choice of meaningful names.

The `logger` module implements the `Logger` type. To access the file system, an object of type `File` (shown in Fig. 2.2) is passed into `logger` as a parameter. The `logger` module's effect

[1]The keyword `resource` in the type definition indicates that the implementations of this type may have state and may access system resources; this is orthogonal to effect checking.

```
resource type Logger
   effect ReadLog
   effect UpdateLog
   def readLog(): {this.ReadLog} String
   def updateLog(newEntry: String): {this.UpdateLog} Unit

module def logger(f: File): Logger
  effect ReadLog = {f.Read}
  effect UpdateLog = {f.Append}
  def readLog(): {this.ReadLog} String = f.read()
  def updateLog(newEntry: String): {this.UpdateLog} Unit = f.append(newEntry
    )
```

Figure 2.1: A type and a module implementing the logging facility in the text-editor application.

```
resource type File
   effect Read
   effect Write
   effect Append
   ...
   def read(): {this.Read} String
   def write(s: String): {this.Write} Unit
   def append(s: String): {this.Append} Unit
   ...
```

Figure 2.2: The type of the file resource.

declarations are those of the `Logger` type, except now they are *concrete*, i.e., they have specific definitions. The `ReadLog` effect of the `logger` module is defined to be the `Read` effect of the `File` object, and accordingly, the `readLog` method, which produces the `ReadLog` effect, calls `f`'s `read` method. Similarly, the `UpdateLog` effect of the `logger` module is defined to be `f.Append`, and accordingly, the `updateLog` method, which produces the `UpdateLog` effect, calls `f`'s `append` method. In general, effects in a module or object definition must always be concrete, whereas effects in a type definition may be either abstract or concrete.

2.1.2 Path-dependent Effects

Effects are members of objects,[2] so we refer to them with the form `variable.EffectName`, where `variable` is an immutable reference to the object defining the effect, and `EffectName` is the name of the effect. For example, in the definition of the `ReadLog` effect of the `logger` module, `f` is the variable referring to a specific file and `Read` is the effect that the `read` method of `f` produces. This conveniently ties together the resource and the effects produced on it (which represent the operations performed on it), helping a software architect or a security analyst to reason about how resources are used by any particular module and its methods. For example, when analyzing the effects produced by `logger`'s `readLog` method, a security analyst can quickly

[2]Modules are an important special case of objects

deduce that calling that method affects the file resource and, specifically, the file is read, simply by looking at the `Logger` type and `logger`'s effect definitions but not at the method's code. Furthermore, these properties can be automatically checked with an idiom of use: In addition to directly looking at the effect annotation of the method of the logger module, the security analyst may write client code that specifies the effect that the logger module is allowed to have. If the logger module accesses system resources outside of the specified effect set, then the compiler would automatically reject the program.

Because an effect includes a reference to an object instance, our effect system can distinguish reads and writes on different file instances. If the developer does not want this level of precision, it is still possible to declare effects at the module level (i.e., as members of a `fileSystem` module object instance), and to share the same `Read` and `Write` effects (for example) across all files in `fileSystem`.

The basic mechanisms of path-dependence are borrowed from Scala and have been shown to scale well in practice. These mechanisms come from the Dependent Object Types (DOT) calculus [1], a type theory of Scala and related languages (including Wyvern). In our system, effects, instead of types are declared as members of objects.

2.1.3 Effect Abstraction

An important and novel feature of our effect system design is the support for *effect abstraction*. Effect abstraction is the ability to define higher-level effects in terms of lower-level effects and potentially to hide that definition from clients of an abstraction. In the logging example above, through the use of abstraction, we "lifted" low-level resources such as the file system (i.e., the `Read` and `Append` effects of the file) into higher-level resources such as a logging facility (i.e., the `ReadLog` and `UpdateLog` effect of the logger) and enabled application code to reason in terms of effects on those higher-level resources when appropriate.

Effect abstraction has several concrete benefits. First, it can be used to distinguish different uses of a low-level effect. For example, `system.FFI` describes any access to system resources via calls through the foreign function interface (FFI), but modules that define file and network I/O can represent these calls as different effects, which enables higher-level modules to reason about file and network access separately. Second, multiple low-level effects can be aggregated into a single high-level effect to reduce effect specification overhead. For instance, the `db.Query` effect might include both `file.Read` and `network.Access` effects. Third, by keeping an effect abstract, we can hide its implementation from clients, which facilitates software evolution: code defining a high-level effect in terms of lower-level ones can be rewritten (or replaced) to use a different set of lower-level effects without affecting clients.

2.1.4 Effect Aggregation

Wyvern's effect-system design allows reducing the effect-annotation overhead by aggregating several effects into one. For example, if, to update the log file, the `logger` module needed to first read the file and then write it back, the `UpdateLog` effect would consist of two effects: a file read and a file write. In other effect systems, this change may make effects more verbose since all the methods that call the `updateLog` method would need to be annotated with the two effects.

However, effect aggregation allows us to define the `UpdateLog` effect to be the two effects and then use `UpdateLog` to annotate the `updateLog` method and all methods that call it:

```
module def logger(f: File): Logger
effect UpdateLog = {f.Read, f.Write}
def updateLog(newEntry: String): {this.UpdateLog} Unit
...
```

This way we need to use only one effect, `UpdateLog`, instead of two, in method effect annotations, thus reducing the effect-annotation overhead. Because more code may add more effects, larger software systems might experience a snowballing of effects, when method annotations have numerous effects in them.

2.1.5 Controlling FFI Effects

Wyvern programs access system resources via calls to other programming languages, such as Java and Python, i.e., through a foreign function interface (FFI). To monitor and control the effects caused by FFI calls, we enforce that all functions from other programming languages, when called within Wyvern, are annotated with the `system.FFI` effect.

As was mentioned in Section 2.1.3, the `system.FFI` effect is an effect that describes function calls through an FFI. Since every function call though FFI has this effect, the access to system resources via FFI is guaranteed to be monitored. `system.FFI` is the lowest-level effect in the effect system which can be used to build other higher-level effects. The programmer can lift `system.FFI` to higher-level effects and reason about those higher-level effects instead.

For example, Wyvern's import mechanism works by loading an object in a static field of a Java class, and the following code imports a field of a Java class that helps to implement file IO:

```
import java:wyvern.stdlib.support.FileIO.file
```

The file object is itself of type FileIO. And FileIO has this method, among others:

```
public void writeStringIntoFile(String content, String filename) throws
    IOException { ... }
```

In Wyvern, there is a type wyvern.stdlib.support.FileIO as well as an object file (of that type) that gets added to the scope as a result of the import above. The type has the following member, corresponding to the method above:

```
def writeStringIntoFile(content:String, filename:String): { system.FFI }
  Unit
```

Here, the system.FFI effect was added to the signature because this is a function that was imported via the FFI. The Wyvern file library that uses the `writeStringIntoFile` function abstracts this `system.FFI` effect into a library-specific `FileIO.Write` effect.

2.1.6 The Limitation of Abstract Effects

In the Wyvern effect system, effects can be declared as members of objects. There are two possible declaration type for an effect member: An effect can be declared abstractly inside a

type: In this example, effect Read is defined as a member of type Logger, the effect Read can be accessed as `v.Read` if `v` has the type `Logger`

```
type Logger
    effect Read
```

On the other hand, an effect can be also defined as an effect set that contains other effects. In the following example, any expression of type `Logger` will have an effect Read that is equivalent to `file.Read`.

```
type Logger
    effect Read = { file.Read }
```

The former type definition for Logger is useful when the programmer wants to hide the definition of the effect Read, while the second type allows the Read effect and `file.Read` to be used interchangeably. However, there is no way for a programmer to have these two benefits at once: the effect Read in Logger is either completely opaque, or completely equivalent to some other effect set.

2.2 Algebraic Effects and Handlers

2.2.1 Overview

Algebraic effects and handlers [25, 26] are a way of implementing certain kinds of side effects such as exceptions and mutable states in an otherwise purely functional setting. As described above, algebraic effects fall into the "denotational" rather than "descriptive" family of work on effects.

Algebraic effects introduce the notion of operations, which carry no predefined meaning. The interpretation for each operation is provided by the evaluation context. In most systems with algebraic effects and handlers, operations are tracked by an effect system, which is similar to the way effectful methods are tracked in the Wyvern programming language. Comparing to restrictive effect systems, algebraic effects subsumes multiple control-flow constructs and can restore the purity of programs by handling effect operations.

Exceptions are an example of control-flow constructs that are subsumed by algebraic effects. Assume that an algebraic effect `exc` is defined with one operation `raise`:

```
effect exc = {
  raise : String -> a
}
```

Then the operation `raise` can be used to implement programs that might cause exceptions. For example, we can write a division function that raises an exception when the divisor is 0.

```
def div(x : Int, y : Int) : {exc} Int
  if (y == 0)
    raise("divide by zero")
  else
    x / y
```

The type of function `div` is $Int \times Int \to \{exc\}\ Int$. The $\{exc\}$ annotation indicates that the function might cause the exc effect by invoking its operation. Because the meaning of operations

is undefined, we need to use a handler to define semantics for effect operations. For this example, we define a handler that returns 0 whenever an error is raised.

```
handle
  div(1, 0)
with
| return x -> x
| raise s -> 0
```

In the return clause, the variable x is bound to the result of the computation if no operation is invoked. So in this case the whole handled expression would evaluate to x. In the `raise` clause, the handler evaluates the whole expression to 0 when an exception is raised.

Another useful property of algebraic effects and handlers is the ability to resume the computation, consider the following handled expression that invokes the function `div` twice.

```
handle
  div(1, 0) + div(2, 1)
with
| return x -> x
| raise s -> resume 3
```

The `raise` clause of the handler is `resume 3` , which tells the program to continue evaluation but use 3 as the result of the operation. So the final result for the whole expression is 5. This example shows that algebraic effects are a more structured form of delimited continuations.

2.2.2 Effect Abstraction

Similar to the restrictive effect system, algebraic effects also benefit from the ability to declare abstract effects. However, this problem is not studied as thoroughly as the effect abstraction in the restrictive setting. The abstraction for algebraic effects was originally proposed by Leijen [13], but was not developed theoretically. The work by Biernacki et al. [2] first studied the abstract algebraic effects formally and proposed a formalization based on coercions. The relationship between Biernacki et al. [2]'s work and ours is discussed more in section 6.1.5

Chapter 3

Bounded Abstract Effects

3.1 Intuition for Subeffecting

Subtyping is used by many programming languages as a means for making the type system flexible. The standard intuition of subtyping is that if every value described by the type `S` is also described by `T`, then `S <: T`. The subtyping relation is also often interpreted as a subset relation, that is, if `S <: T`, then the elements of `S` are a subset of the elements of `T`. Type systems incorporates subtyping relation by adding the rule of subsumption:

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

which states that if a term `t` belongs to a type `S`, and `S` is a subtype if `T`, then `t` can be used as a value of type `T`.

There is a natural way to incorporate the interpretation of subtyping into the Wyvern effect system. Since the effect system describes the set of side effects a program can possibly cause, if one effect `E` is a subset of another effect `F`, then it is safe to use a term that has effect `E` whenever a term of effect `F` is expected. We therefore consider the effect `E` as a subeffect of the effect `F`, due to their similarity to subtyping. To give a concrete example, consider a declaration of a method `readFromNetwork`:

```
def readFromNetwork() : {network.Read, file.Write} Unit = writeFile()
def writeFile() : {file.Write} Unit = file.Write{"something"}
```

The `readFromNetwork` method has an effect that consists of two effect labels, `network.Read` and `file.Write`, and is implemented by calling `writeFile` method, which only has the `file.Write` effect. Because the effect of `writeFile` method is a subset of the effect of `readFromNetwork`, it can be safely used as an implementation of `readFromNetwork`.

3.2 Effect Bounds

Our effect system gives the programmer the ability to define a subtyping hierarchy of effects via effect bounds. To define the hierarchy, the programmer gives the effect member an upper bound or a lower bound, hiding the definition of the effect from the client.

For example, consider the type `BoundedLogger` which has the same method declarations and effect members as the type `Logger` in Fig. 2.1, except the `ReadLog` and `UpdateLog` effects are upper-bounded by the corresponding effects in the `fileSystem` module:

```
type BoundedLogger
   effect ReadLog <= {fileSystem.Read}
   effect UpdateLog <= {fileSystem.Append}
   ...
```

Any object implementing the type `BoundedLogger` may have an effect member `ReadLog` which is *at most* `fileSystem.Read`. This allows programmers to compare the `ReadLog` effect with other effects, while keeping its definition abstract. For instance, a library can provide two implementations of `BoundedLogger`, including an effectless logger in which the effects `ReadLog` and `UpdateLog` are empty sets, and an effectful logger in which `ReadLog` and `UpdateLog` are defined as effects in the `fileSystem` module. The library's clients then can annotate the effects of both implementations with `fileSystem.Read` and `fileSystem.Append` according to the effect hierarchy, without the need to know the exact implementation of the two instances.

Effect hierarchy can also be constructed using lower bounds. For example, consider the following type for I/O modules that supports writes:

```
type IO
  effect Write >= {system.FFI}
  def write(s: String): {this.Write} Unit
```

Since I/O is done using the foreign function interface (FFI), the `Write` effect is *at least* the `system.FFI` effect. Similar to providing an upper bounded on effects, this type does not specify the exact definition of the `Write` effect, and implementations of this type can define `Write` as an effect set with more effects than `{system.FFI}`.

The effect hierarchy achieved by bounding effect members is supported by the subtyping relations of our effect system (Sections 3.3.5). If a type has an effect member with more strict bounds than another type, then the former type is a subtype of the latter type. For example, when a logger with the effect member `Read <= {fileSystem.Read}` is expected, we can pass in a logger with `Read = {}` because the definition as an empty set is more strict than an upper bound.

The following two case studies demonstrates the expressiveness of the effect hierarchy:

3.2.1 Controlling Access to UI Objects

This main idea of the work of Gordon et al. [8] is to control the access of user interface (UI) framework methods so that unsafe UI methods can only be called on the UI thread. There are three different method annotations `@SafeEffect`, `@UIEffect`, and `@PolyUIEffect`, where

1. `@SafeEffect` annotates methods that are safe to run on any thread,
2. `@UIEffect` annotates methods that is only callable on UI thread, and
3. `@PolyUIEffect` annotates methods whose effect is polymorphic over the receiver type's effect parameter.

In Wyvern, we can model `@UIEffect` as a member of the UI module, for example:

```
type UILibrary
  effect UIEffect >= {system.FFI}
```

```
  def unsafeUIMethod1(): {this.UIEffect} Unit
  def unsafeUIMethod2(): {this.UIEffect} Unit
  ...
```

This way, any client code of an UI library that calls UI methods will have the `uilibrary.UIEffect` effect.

An interface could be used for UI-effectful or UI-safe work. To accommodate such flexibility, $Java_{UI}$ introduced the `@PolyUIType` annotation. For example, a `Runnable` interface which can be UI-safe or UI-unsafe is declared as

```
@PolyUIType public interface Runnable {
  @PolyUIEffect void Run();
}
```

Whether the method `Run()` will have a UI effect depends on an annotation when the type is instantiated. For example:

```
@Safe Runnable s =....;
s.run(); // is UI safe
@UI Runnable s = .....;
s.run(); // has UI effect
```

In Wyvern, such polymorphic interface can be created by defining the interface with a bounded effect member:

```
type Runnable
  effect Run <= {uiLibrary.UIEffect}
  def run(): {this.Run} Unit
```

This type ensures that the `run` method is safe to be called on the UI thread. Moreover, if an instance of `Runnable` does not have `UIEffect`, it can be ascribed with the type `SafeRunnable`, which is a subtype of `Runnable`:

```
type SafeRunnable
  effect Run = {}
  def run(): {this.Run} Unit
```

This indicates that `run` is safe to be called on any thread.

3.2.2 Controlling Mutable States Using Abstract Regions

Greenhouse and Boyland [9] proposed a region-based effect system which describes how state may be accessed during the execution of some program component in object-oriented programming languages. One example of the usage of regions is as follows:

```
class Point {
  public region Position;
  private int x in Position;
  private int y in Position;
  public scale(int sc) reads nothing writes Position {
    x *= sc;
    y *= sc;
  }
}
```

The two variables `x` and `y` are declared inside a region `Position`. For each region, there can be two possible effects: read and write. The `scale` method has the effect of writing on the region `this.Position`.

To achieve access control on regions in Wyvern, we need to keep track of the read and write effect on each variable in a region. We declare the resource type `Var` representing a variable wrapper.

```
resource type Var[T]
  effect Read
  effect Write
  def set (x: T): {this.Write} Unit
  def get (): {this.Read} T
```

Since the `set` and `get` methods are annotated with the corresponding effects and there is no exposed access to the variable that holds the value, the two methods protect the access to the variable inside the type `Var`. To avoid code boilerplate, this wrapper type can be added as a language extension. The `Point` example above can be rewritten in Wyvern as:

```
resource type Point
  val x: Var[Int]
  val y: Var[Int]
  effect Read >= {this.x.Read, this.y.Read}
  effect Write >= {this.x.Write, this.y.Write}
  def scale(sc: Int): {this.Write} Unit
```

We can also extend the type `Point` to `3DPoint` in the following way:

```
resource type 3DPoint
  val x: Var[Int]
  val y: Var[Int]
  val z: Var[Int]
  effect Read = {this.x.Read, this.y.Read, this.z.Read}
  effect Write = {this.x.Write, this.y.Write, this.z.Write}
  def scale(sc: Int): {this.Write} Unit
```

Since the effect `Read` and `Write` in the type `Point` is declared with a lower bound, the type `3DPoint` is a subtype of `Point`.

3.3 Formalization

As was mentioned earlier, Wyvern modules are first class and are, in fact, objects since they are only syntactic sugar on top of Wyvern's object-oriented core and can be translated into objects. The translation has been described in detail previously [18], and here we provide only some intuition behind it. In this section, we start with describing the syntax of Wyvern's object-oriented core, then present an example of the module-to-object translation, followed by a description of Wyvern's static semantics and subtyping rules. Furthermore, we present the dynamic semantics and the type soundness theorems. Last but not least, we provide the definitions on authority and discuss why they are useful for security analysis on programs written in Wyvern.

3.3.1 Object-Oriented Core Syntax

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \texttt{new}\ (x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & \texttt{def}\ m(x : \tau) : \{\varepsilon\}\ \tau = e \\
 & | & \texttt{var}\ f : \tau = x \\
 & | & \texttt{effect}\ g = \{\varepsilon\} \\
\varepsilon & ::= & \overline{x.g} \\
\tau & ::= & \{x \Rightarrow \overline{\sigma}\} \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x : \tau
\end{array}
\qquad
\begin{array}{rcl}
\sigma & ::= & \texttt{def}\ m(x : \tau) : \{\varepsilon\}\ \tau \\
 & | & \texttt{var}\ f : \tau \\
 & | & \texttt{effect}\ g \\
 & | & \texttt{effect}\ g \geqslant \{\varepsilon\} \\
 & | & \texttt{effect}\ g \leqslant \{\varepsilon\} \\
 & | & \texttt{effect}\ g = \{\varepsilon\}
\end{array}
$$

Figure 3.1: Wyvern's object-oriented core syntax.

Fig. 3.1 shows the syntax of Wyvern's object-oriented core. Wyvern expressions include variables and the four basic object-oriented expressions: the `new` statement, a method call, a field access, and a field assignment. Objects are created by `new` statements that contain a variable x representing the current object along with a list of declarations. In our implementation, x defaults to `this` when no name is specified by the programmer. Declarations come in three kinds: a method declaration, a field, and an effect member. Method declarations are annotated with a set of effects. Object fields may only be initialized using variables, a restriction which simplifies our core language by ensuring that object initialization never has an effect. Although at first this may seem to be limiting, in fact, we do not limit the source language in this way. Side-effecting member initializations in the source language are translated to the core by wrapping the new object with a `let` expression (a discussion of which is upcoming) that defines the variable to be used in the field initialization. For example, this code:

```
new
   var x: String = f.read()
```

can be internally rewritten as:

```
let y = f.read()
in new
   var x: String = y
```

Effects in method annotations and effect-member definitions are surrounded by curly braces to visually indicate that they are sets, and each effect in an effect set is defined to be a variable representing the object on which an effect is produced, followed by a dot and the effect name. Abstract effects may be defined with an upper bound or a lower bound.

Object types are a collection of declaration types, which include method signatures, field-declaration types, and the types of effect-member declarations and definitions. Similar to the difference between the modules and their types, effects in an object must always be defined (i.e., always be concrete), whereas effects in object types may or may not have definitions (i.e., be either abstract or concrete), and may have an upper or lower bound.

3.3.2 Modules-to-Objects Translation

Fig. 3.2 presents a simplified translation of the `logger` module from Fig. 2.1 into Wyvern's object-oriented core (for a full description of the translation mechanism, refer to [18]). For our

```
1 let logger = new(x ⇒
2   def apply(f : File) : {} Logger
3       new(_ ⇒
4         effect ReadLog = {f.Read}
5         effect UpdateLog = {f.Append}
6         def readLog() : {ReadLog} String = f.read()
7         def updateLog(newEntry : String) : {UpdateLog} Unit = f.append(newEntry)))
8 in ...// calls logger.apply(...)
```

Figure 3.2: A simplified translation of the `logger` module from Fig. 2.1 into Wyvern's object-oriented core.

purposes, the functor becomes a regular method, called `apply`, that has the return type `Logger` and the same parameters as the module functor. The method's body is a new object containing all the module declarations. The `apply` method is the only method of an outer object that is assigned to a variable whose name is the module's name. Later on in the code, when the `logger` module needs to be instantiated, the `apply` method is called with appropriate arguments passed in.

To aid this translation mechanism, we use the two relatively standard encodings:

$$\texttt{let}\ x = e\ \texttt{in}\ e' \equiv \texttt{new}(\_ \Rightarrow \texttt{def}\ f(x : \tau) : \tau' = e').f(e)$$

$$\texttt{def}\ m(\overline{x : \tau}) : \tau = e \equiv \texttt{def}\ m(x : (\tau_1 \times \tau_2 \times ... \times \tau_n)) : \tau = [x.n/x_n]e$$

The `let` expression is encoded as a method call on an object that contains that method with the `let` variable being the method's parameter and the method body being the `let`'s body. The multiparameter version of the method definition is encoded using indexing into the method parameters.

3.3.3 Well-formedness

Since Wyvern's effects are defined in terms of variables, before we typecheck expressions, we must make sure that effects and types are well formed. Wyvern well-formedness rules are mostly straightforward and are shown in Fig. 3.3. The three judgements read that, in the variable typing context Γ, the type τ, the declaration type σ, and the effect set ε are well formed, respectively.

An object type is well formed if all of its declaration types are well formed. A method-declaration type is well formed if the type of its parameter, its return type, and the effects in its effect annotation are well formed. A field-declaration type is well formed if its type is well formed. Since an effect-declaration type has no right-hand side, it is trivially well formed. An effect-definition type is well formed if the effect set in its right-hand side is well formed. Finally, a bounded effect declaration is well formed if the upper bound or lower bound on the right-hand side is well formed. An effect set is well formed if, for every effect it contains, the definition of the effect doesn't form a cycle, the variable in the first part of the effect is well typed and the type of that variable contains either an effect-declaration or an effect-definition type, in which the effect name matches the effect name in the second part of the effect.

The $\Gamma \vdash safe(x.g, \varepsilon)$ judgment ensures that the definition of effect $x.g$ doesn't contain a cycle. The rules Safe-1, Safe-2, and Safe-3 are identical except the declaration of the effect type.

$\boxed{\Gamma \vdash \tau \ wf}$

$$\frac{\forall \sigma \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \vdash \sigma \ wf}{\Gamma \vdash \{x \Rightarrow \overline{\sigma}\} \ wf} \text{ (WF-TYPE)}$$

$\boxed{\Gamma \vdash \sigma \ wf}$

$$\frac{\Gamma \vdash \tau_2 \ wf \quad \Gamma,\ x : \tau_2 \vdash \tau_1 \ wf \quad \Gamma,\ x : \tau_2 \vdash \varepsilon \ wf}{\Gamma \vdash \texttt{def}\ \ m(x : \tau_2) : \{\varepsilon\}\ \tau_1 \ wf} \text{ (WF-DEF)} \qquad \frac{\Gamma \vdash \tau \ wf}{\Gamma \vdash \texttt{var}\ f : \tau \ wf} \text{ (WF-VAR)}$$

$$\frac{}{\Gamma \vdash \texttt{effect}\ g \ wf} \text{ (WF-EFFECT1)} \qquad \frac{\Gamma \vdash \varepsilon \ wf}{\Gamma \vdash \texttt{effect}\ g = \{\varepsilon\} \ wf} \text{ (WF-EFFECT2)}$$

$$\frac{\Gamma \vdash \varepsilon \ wf}{\Gamma \vdash \texttt{effect}\ g \leqslant \{\varepsilon\} \ wf} \text{ (WF-EFFECT3)} \qquad \frac{\Gamma \vdash \varepsilon \ wf}{\Gamma \vdash \texttt{effect}\ g \geqslant \{\varepsilon\} \ wf} \text{ (WF-EFFECT4)}$$

$\boxed{\Gamma \vdash \varepsilon \ wf}$

$$\frac{\begin{array}{c}\forall i, j,\ x_i.g_j \in \varepsilon,\ \Gamma \vdash safe(x_i.g_j, \{\}),\ \Gamma \vdash x_i : \{\}\ \{y_i \Rightarrow \overline{\sigma_i}\}, \\ (\texttt{effect}\ g_j \in \overline{\sigma_i} \vee \texttt{effect}\ g_j = \{\varepsilon_j\} \in \overline{\sigma_i} \vee \texttt{effect}\ g_j \geqslant \{\varepsilon_j\} \in \overline{\sigma_i} \vee \texttt{effect}\ g_j \leqslant \{\varepsilon_j\} \in \overline{\sigma_i})\end{array}}{\Gamma \vdash \varepsilon \ wf} \text{ (WF-EFFECT)}$$

$\boxed{\Gamma \vdash safe(x.g, \varepsilon)}$

$$\frac{\begin{array}{l}\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \texttt{effect}\ g = \{\varepsilon'\} \in \overline{\sigma} \\ \forall a.b \in \{x.g\} \cup \varepsilon, a.b \notin [x/y]\varepsilon' \\ \forall c.d \in [x/y]\varepsilon', \Gamma \vdash safe(c.d, \{x.g\} \cup \varepsilon)\end{array}}{\Gamma \vdash safe(x.g, \varepsilon)} \text{ (SAFE-1)}$$

$$\frac{\begin{array}{l}\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \texttt{effect}\ g \geqslant \{\varepsilon'\} \in \overline{\sigma} \\ \forall a.b \in \{x.g\} \cup \varepsilon, a.b \notin [x/y]\varepsilon' \\ \forall c.d \in [x/y]\varepsilon', \Gamma \vdash safe(c.d, \{x.g\} \cup \varepsilon)\end{array}}{\Gamma \vdash safe(x.g, \varepsilon)} \text{ (SAFE-2)}$$

$$\frac{\begin{array}{l}\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \texttt{effect}\ g \leqslant \{\varepsilon'\} \in \overline{\sigma} \\ \forall a.b \in \{x.g\} \cup \varepsilon, a.b \notin [x/y]\varepsilon' \\ \forall c.d \in [x/y]\varepsilon', \Gamma \vdash safe(c.d, \{x.g\} \cup \varepsilon)\end{array}}{\Gamma \vdash safe(x.g, \varepsilon)} \text{ (SAFE-3)}$$

$$\frac{\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \texttt{effect}\ g}{\Gamma \vdash safe(x.g, \varepsilon)} \text{ (SAFE-4)}$$

Figure 3.3: Wyvern well-formedness rules.

The effect set ε memorizes a set of effects that are defined by $x.g$. The rule ensures that those effects do not appear in the definition of $x.g$, therefore eliminating cycles in effect definition.

$\boxed{\Gamma \vdash e : \{\varepsilon\}\ \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \{\}\ \tau}\ \text{(T-VAR)} \qquad \frac{\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \vdash d_i : \sigma_i}{\Gamma \vdash \texttt{new}(x \Rightarrow \overline{d}) : \{\}\ \{x \Rightarrow \overline{\sigma}\}}\ \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \{\varepsilon_1\}\{x \Rightarrow \overline{\sigma}\} \quad \texttt{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma} \\ \Gamma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf \quad \Gamma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2 \quad \varepsilon = \varepsilon_1 \cup \varepsilon_2 \cup [e_1/x][e_2/y]\varepsilon_3\end{array}}{\Gamma \vdash e_1.m(e_2) : \{\varepsilon\}\ [e_1/x][e_2/y]\tau_1}\ \text{(T-METHOD)}$$

$$\frac{\Gamma \vdash e : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\} \quad \texttt{var}\ f : \tau \in \overline{\sigma}}{\Gamma \vdash e.f : \{\varepsilon\}\ [e/x]\tau}\ \text{(T-FIELD)} \qquad \frac{\Gamma \vdash e : \{\varepsilon_1\}\ \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash e : \{\varepsilon_2\}\ \tau_2}\ \text{(T-SUB)}$$

$$\frac{\Gamma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\} \quad \texttt{var}\ f : \tau \in \overline{\sigma} \quad \Gamma \vdash e_2 : \{\varepsilon_2\}\ \tau \quad \varepsilon = \varepsilon_1 \cup \varepsilon_2}{\Gamma \vdash e_1.f = e_2 : \{\varepsilon\}\ [e_1/x]\tau}\ \text{(T-ASSIGN)}$$

$\boxed{\Gamma \vdash d : \sigma}$

$$\frac{\begin{array}{c}\Gamma,\ x : \tau_1 \vdash e : \{\varepsilon_2\}\ \tau_2 \quad \Gamma,\ x : \tau_1 \vdash \varepsilon_1\ wf \\ \Gamma, x : \tau_1 \vdash \varepsilon_2 <: \varepsilon_1\end{array}}{\Gamma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 = e\ :\ \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2}\ \text{(DT-DEF)}$$

$$\frac{\Gamma \vdash x : \{\}\ \tau}{\Gamma \vdash \texttt{var}\ f : \tau = x\ :\ \texttt{var}\ f : \tau}\ \text{(DT-VAR)} \qquad \frac{\Gamma \vdash \varepsilon\ wf}{\Gamma \vdash \texttt{effect}\ g = \{\varepsilon\}\ :\ \texttt{effect}\ g = \{\varepsilon\}}\ \text{(DT-EFFECT)}$$

Figure 3.4: Wyvern static semantics.

3.3.4 Static Semantics

Wyvern's static semantics is presented in Fig. 3.4. Expression type checking includes checking the effects that an expression may have, the set of which is denoted in a pair of curly braces between the colon and the type in the type annotation. Then, for expressions, the judgement reads that, in the variable typing context Γ, the expression e is a well-typed expression with the effect set ε and the type τ.

A variable trivially has no effects. A `new` expression also has no effects because of the fact that fields may be initialized only using variables. A new object is well typed if all of its declarations are well typed.

A method call is well typed if the expression passed into the method as an argument is well typed, if the expression the method is called on is well typed, and if the expression's type contains a matching method-declaration type. In addition, bearing the appropriate variable substitutions, the effect set annotating the method-declaration type must be well formed, and the effect set ε in the method-call type must be a union of the effect sets of both expressions involved in the method call as well as the the effect set of the method-declaration type. The expressions that are being substituted are always the terminal runtime form, i.e., the expressions have been fully evaluated before they are substituted.

An object field read is well typed if the expression on which the field is dereferenced is well

typed and the expression's type contains a matching field-declaration type. The effects of an object field type are those of the expression on which the field dereferencing is called.

A field assignment is well typed if the expression to which the field belongs is well typed and the expression's type has an appropriate field-declaration type, and if the expression in the right-hand side of the assignment is well typed. The effect set that a field assignment produces is a union between the effect sets the two expressions that are involved in the field assignment produce.

A type substitution of an expression may happen only if the expression is well typed using the original type, the original type is a subtype of the new type, and when the effect set of the original set is a subeffect of the effect of the new type. (Subeffecting is discussed in Section 3.3.5.)

None of the object declarations produce effects, and so object-declaration type-checking rules do not include an effect set preceding the type annotation. For declarations, the judgement reads that, in the variable typing context Γ, the declaration d is a well-typed declaration with the type σ.

When type-checking a method declaration, the effect set annotating the method must be well formed in the overall typing context extended with the method argument. Furthermore, the effect annotating the method must be a supereffect of the effect the method body actually produced.

A field declaration is trivially well typed, and an effect declaration is well typed if the effect set that it is defined with is well formed in the given context.

3.3.5 Subtyping

Subeffecting Rules

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

$$\frac{\varepsilon_1 \subseteq \varepsilon_2}{\Gamma \vdash \varepsilon_1 <: \varepsilon_2} \text{ (SUBEFFECT-SUBSET)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g \leqslant \varepsilon \in \sigma \quad \Gamma \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2} \text{ (SUBEFFECT-UPPERBOUND)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g \geqslant \{\varepsilon\} \in \sigma \quad \Gamma \vdash \varepsilon_1 <: [n/y]\varepsilon \cup \varepsilon_2}{\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{n.g\}} \text{ (SUBEFFECT-LOWERBOUND)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g = \{\varepsilon\} \in \sigma \quad \Gamma \vdash \varepsilon_1 <: [n/y]\varepsilon \cup \varepsilon_2}{\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{n.g\}} \text{ (SUBEFFECT-DEF-1)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g = \{\varepsilon\} \in \sigma \quad \Gamma \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2} \text{ (SUBEFFECT-DEF-2)}$$

Figure 3.5: Wyvern subeffecting rules.

As we already saw in the T-SUB, and DT-DEF rules above, to compare two sets of effects, we use subeffecting rules, which are presented in Fig. 3.5. If an effect is a subset of another effect,

$\boxed{size(\Gamma, \varepsilon) = n}$

$$\frac{}{size(\Gamma, \{\}) = 0} \text{ (SIZE-EMPTY)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g \in \sigma}{size(\Gamma, x.g) = 0} \text{ (SIZE-ABSTRACT)}$$

$$\frac{}{size(\Gamma, \overline{x.g}) = \Sigma_{x.g \in \overline{x.g}} size(\Gamma, x.g)} \text{ (SIZE-LIST)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g = \{\varepsilon\} \in \sigma}{size(\Gamma, x.g) = 1 + size(\Gamma, [x/y]\varepsilon)} \text{ (SIZE-DEF)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g \leqslant \{\varepsilon\} \in \sigma}{size(\Gamma, x.g) = 1 + size(\Gamma, [x/y]\varepsilon)} \text{ (SIZE-UPPERBOUND)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g \geqslant \{\varepsilon\} \in \sigma}{size(\Gamma, x.g) = 1 + size(\Gamma, [x/y]\varepsilon)} \text{ (SIZE-LOWERBOUND)}$$

Figure 3.6: Rules for determining the size of effect definitions.

then the former effect is a subeffect of the latter (SUBEFFECT-SUBSET). If an effect set contains an effect variable that is declared with an upper bound, and the union of the rest of the effect set with the upper bound is a subeffect of another effect set, then the former effect set is a subeffect of the latter effect set (SUBEFFECT-LOWERBOUND). If an effect set contains an effect variable that is declared with an lower bound, and the union of the rest of the effect set with the lower bound is a supereffect of another effect set, then the former effect set is a supereffect of the latter (SUBEFFECT-LOWERBOUND). If an effect set contains an effect variable that has a definition, and the union of the rest of the effect set with the definition of the variable is a supereffect of another effect set, then the former effect set is a supereffect of the latter (SUBEFFECT-DEF-1). Finally, if an effect set contains an effect variable that has a definition, and the union of the rest of the effect set with the definition of the variable is a subeffect of another effect set, then the former effect set is a subeffect of the latter (SUBEFFECT-DEF-2).

Lemma 1. $size(\Gamma, \varepsilon)$ (Defined in Fig. 3.6) is finite.

Proof. By rules Safe-1, Safe-2, Safe-3, and Safe-4 in Fig. 3.3, the size of an arbitrary effect $x.g$ is bounded by the total number of effects in the context Γ. □

Theorem 2. $\Gamma \vdash \varepsilon <: \varepsilon'$ is decidable.

Proof. The proof is by induction on $size(\Gamma, \varepsilon \cup \varepsilon')$.

- BC Since size for both effect is 0, the only applicable rule for subeffecting is Subeffect-Subset. The rule only checks if ε is a subset of ε', therefore is decidable.
- IS Assume the judgment $\Gamma \vdash \varepsilon <: \varepsilon'$ is derived from Subeffect-Upperbound. In the premise of this rule, we have $\Gamma \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2$. Since we extract the definition of $n.g$ to find ε,

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

$$\frac{}{\Gamma \vdash \tau <: \tau} \text{ (S-REFL1)} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \text{ (S-TRANS)}$$

$$\frac{\{x \Rightarrow \sigma_i^{i\in 1..n}\} \textit{ is a permutation of } \{x \Rightarrow \sigma_i'^{i\in 1..n}\}}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1..n}\} <: \{x \Rightarrow \sigma_i'^{i\in 1..n}\}} \text{ (S-PERM)}$$

$$\frac{}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1..n+k}\} <: \{x \Rightarrow \sigma_i^{i\in 1..n}\}} \text{ (S-WIDTH)} \quad \frac{\forall i,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..n}\} \vdash \sigma_i <: \sigma_i'}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1..n}\} <: \{x \Rightarrow \sigma'_i{}^{i\in 1..n}\}} \text{ (S-DEPTH)}$$

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

$$\frac{}{\Gamma \vdash \sigma <: \sigma} \text{ (S-REFL2)} \quad \frac{\Gamma \vdash \tau_1' <: \tau_1 \quad \Gamma \vdash \tau_2 <: \tau_2' \quad \Gamma, x : \tau_1 \vdash \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 <: \texttt{def}\ m(x : \tau_1') : \{\varepsilon_2\}\ \tau_2'} \text{ (S-DEF)}$$

$$\frac{}{\Gamma \vdash \texttt{effect}\ g = \{\varepsilon\} <: \texttt{effect}\ g} \text{ (S-EFFECT-1)} \quad \frac{}{\Gamma \vdash \texttt{effect}\ g \leqslant \varepsilon <: \texttt{effect}\ g} \text{ (S-EFFECT-2)}$$

$$\frac{\Gamma \vdash \varepsilon <: \varepsilon'}{\Gamma \vdash \texttt{effect}\ g = \{\varepsilon\} <: \texttt{effect}\ g \leqslant \varepsilon'} \text{ (S-EFFECT-3)} \quad \frac{\Gamma \vdash \varepsilon <: \varepsilon'}{\Gamma \vdash \texttt{effect}\ g \leqslant \varepsilon <: \texttt{effect}\ g \leqslant \varepsilon'} \text{ (S-EFFECT-4)}$$

$$\frac{}{\Gamma \vdash \texttt{effect}\ g \geqslant \varepsilon <: \texttt{effect}\ g} \text{ (S-EFFECT-5)} \quad \frac{\Gamma \vdash \varepsilon' <: \varepsilon}{\Gamma \vdash \texttt{effect}\ g = \{\varepsilon\} <: \texttt{effect}\ g \geqslant \varepsilon'} \text{ (S-EFFECT-6)}$$

$$\frac{\Gamma \vdash \varepsilon' <: \varepsilon}{\Gamma \vdash \texttt{effect}\ g \geqslant \varepsilon <: \texttt{effect}\ g \geqslant \varepsilon'} \text{ (S-EFFECT-7)}$$

Figure 3.7: Wyvern subtyping rules.

we have $size(\Gamma, [n/y]\varepsilon \cup \varepsilon_1 \cup \varepsilon_2) < size(\Gamma, \{n.g\} \cup \varepsilon_1 \cup \varepsilon_2)$. We can then use induction hypothesis to show the subeffecting judgment in the premise is decidable.
The inductive step for rules Subeffect-Lowerbound, Subeffect-Def-1, and Subeffect-Def-2 have the similar structure.

□

Declarative Subtyping Rules

Wyvern subtyping rules are shown in Fig. 3.7. Since, to compare types, we need to compare the effects in them using subeffecting, subtyping relationship is checked in a particular variable typing context. The first four object-subtyping rules and the S-REFL2 rule are standard. In S-DEPTH, since effects may contain a reference to the current object, to check the subtyping relationship between two type declarations, we extend the current typing context with the current object. Method-declaration typing is contravariant in the argument types and covariant in the return type. Furthermore, there must be a covariant-like relationship between the effect sets

in the method annotations on the two method declarations: the effect set of the subtype method declaration must be a subeffect of the effect set of the supertype method declaration (S-DEF). An effect definition or an effect declaration with bound is trivially a subtype of an effect declaration (S-EFFECT-1, S-EFFECT-2, S-EFFECT-5). An effect definition is a subtype of an effect declaration with upper bound if the definition is a subeffect of the upper bound (S-EFFECT-3). Similarly, an effect definition is a subtype of an effect declaration with lower bound if the definition is a supereffect of the lower bound (S-EFFECT-6). An effect declaration with upper bound is a subtype of the effect declaration with another upper bound if the former upper bound is a subeffect of the latter upper bound (S-EFFECT-4). Finally, an effect declaration with lower bound is a subtype of the effect declaration with another lower bound if the former upper bound is a supereffect of the latter upper bound (S-EFFECT-7).

Algorithmic Subtyping Rules

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

$$\frac{\exists \textit{ an injection } p : \{1...n\} \mapsto \{1...m\}, \quad \forall i \in 1...n,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i \in 1..m}\} \vdash \sigma_{p(i)} <: \sigma'_i}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i \in 1...m}\} <: \Gamma \vdash \{x \Rightarrow {\sigma'}_i^{i \in 1...n}\}} \text{ (S-ALG)}$$

Figure 3.8: Algorithmic Subtyping

The S-Alg rule encodes the S-Refl-1, S-Perm, S-Depth, and S-Width rule using an injective function p. The subtyping rules of declaration types are identical to the declarative subtyping. We prove that S-Trans rules is admissible in theorem 3. Since subtyping rules object types and declaration types are syntax-directed, the subtyping of our effect system is decidable.

Theorem 3. (Transitivity of algorithmic subtyping)
If $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, then $\Gamma \vdash \tau_1 <: \tau_3$.
If $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$, then $\Gamma \vdash \sigma_1 <: \sigma_3$.

3.3.6 Dynamic Semantics and Type Soundness

Object-Oriented Core Syntax

Fig. 3.9 shows the version of the syntax of Wyvern's object-oriented core that includes dynamic semantics. Specifically, expressions include locations l, which variables in effects resolve to at run time. We also use a store μ and its typing context Σ. Finally, to make the dynamics more compact we use an evaluation context E.

Changes in Static Semantics

Type checking a location (T-LOC) and a field declaration (DT-VAR) is straightforward, and we also need to ensure that the store is well-formed and contains objects that respect their types.

$$
\begin{array}{rcll}
n & ::= & x \mid l & \textit{names} \\
e & ::= & n & \textit{expressions} \\
 & \mid & \texttt{new}(x \Rightarrow \overline{d}) & \\
 & \mid & e.m(e) & \\
 & \mid & e.f & \\
 & \mid & e.f = e & \\
\varepsilon & ::= & \overline{n.g} & \textit{effects} \\
d & ::= & \texttt{def}\ m(x:\tau):\{\varepsilon\}\ \tau = e & \textit{declarations} \\
 & \mid & \texttt{var}\ f:\tau = n & \\
 & \mid & \texttt{effect}\ g = \{\varepsilon\} & \\
\tau & ::= & \{x \Rightarrow \overline{\sigma}\} & \textit{object type}
\end{array}
\qquad
\begin{array}{rcll}
\sigma & ::= & \texttt{def}\ m(x:\tau):\{\varepsilon\}\ \tau & \textit{declaration types} \\
 & \mid & \texttt{var}\ f:\tau & \\
 & \mid & \texttt{effect}\ g & \\
 & \mid & \texttt{effect}\ g \geqslant \{\varepsilon\} & \\
 & \mid & \texttt{effect}\ g \leqslant \{\varepsilon\} & \\
 & \mid & \texttt{effect}\ g = \{\varepsilon\} & \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x:\tau & \textit{var. typing context} \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{x \Rightarrow \overline{d}\} & \textit{store} \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l:\tau & \textit{store typing context} \\
E & ::= & [\,] & \textit{evaluation context} \\
 & \mid & E.m(e) & \\
 & \mid & l.m(E) & \\
 & \mid & E.f & \\
 & \mid & E.f = e & \\
 & \mid & l.f = E &
\end{array}
$$

Figure 3.9: Wyvern's object-oriented core syntax with dynamic forms.

$\boxed{\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau}$

$$\ldots \quad \frac{l:\tau \in \Sigma}{\Gamma \mid \Sigma \vdash l : \{\}\ \tau}\ \text{(T-Loc)}$$

$\boxed{\Gamma \mid \Sigma \vdash d : \sigma}$

$$\ldots \quad \frac{\Gamma \mid \Sigma \vdash n : \{\}\ \tau}{\Gamma \mid \Sigma \vdash \texttt{var}\ f:\tau = n\ :\ \texttt{var}\ f:\tau}\ \text{(DT-Var)}$$

$\boxed{\mu : \Sigma}$

$$\frac{\forall l \mapsto \{x \Rightarrow \overline{d}\} \in \mu,\ \forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ x : \{x \Rightarrow \overline{\sigma}\} \mid \Sigma \vdash d_i : \sigma_i}{\mu : \Sigma}\ \text{(T-Store)}$$

Figure 3.10: Wyvern static semantics affected by dynamic semantics.

Dynamic Semantics

The dynamic semantics that we use for Wyvern's effect system is shown in Fig. 3.11 and is similar to the one described in prior work [18]. In comparison to the prior work, this version of Wyvern's dynamic semantics has fewer rules, and the E-Method rule is simplified.

The judgement reads the same as before: given the store μ, the expression e evaluates to the expression e' and the store becomes μ'. The E-Congruence rule still handles all non-terminal forms. To create a new object (E-New), we select a fresh location in the store and assign the object's definition to it. Provided that there is an appropriate method definition in the object on which a method is called, the method call is reduced to the method's body (E-Method). In the method's body, the locations representing the method argument and the object on which the method is called are substituted for corresponding variables. An object field is reduced to the value held in it (E-Field), and when an object field's value changes (E-Assign), appropriate

$\boxed{\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle}$

$$\frac{\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle}{\langle E[e] \mid \mu \rangle \longrightarrow \langle E[e'] \mid \mu' \rangle}\ \text{(E-CONGRUENCE)} \qquad \frac{l \notin dom(\mu)}{\langle \texttt{new}(x \Rightarrow \overline{d}) \mid \mu \rangle \longrightarrow \langle l \mid \mu, l \mapsto \{x \Rightarrow \overline{d}\} \rangle}\ \text{(E-NEW)}$$

$$\frac{l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu \quad \texttt{def}\ m(y : \tau_1) : \{\varepsilon\}\ \tau_2 = e \in \overline{d}}{\langle l_1.m(l_2) \mid \mu \rangle \longrightarrow \langle [l_2/y][l_1/x]e \mid \mu \rangle}\ \text{(E-METHOD)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\} \in \mu \quad \texttt{var}\ f : \tau = l_1 \in \overline{d}}{\langle l.f \mid \mu \rangle \longrightarrow \langle l_1 \mid \mu \rangle}\ \text{(E-FIELD)}$$

$$\frac{\begin{array}{c} l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu \qquad \texttt{var}\ f : \tau = l \in \overline{d} \\ \overline{d}' = [\texttt{var}\ f : \tau = l_2/\texttt{var}\ f : \tau = l]\overline{d} \quad \mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}/l_1 \mapsto \{x \Rightarrow \overline{d}\}]\mu \end{array}}{\langle l_1.f = l_2 \mid \mu \rangle \longrightarrow \langle l_2 \mid \mu' \rangle}\ \text{(E-ASSIGN)}$$

Figure 3.11: Wyvern dynamic semantics.

substitutions are made in the object's declaration set and the store.

Type Soundness

We prove the soundness of the effect system presented above using the standard combination of progress and preservation theorems. Proof to these theorems can be found in Appendix B.

Theorem 4 (Preservation)**.** If $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, $\mu : \Sigma$, and $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$, then $\exists \Sigma' \supseteq \Sigma$, $\mu' : \Sigma'$, $\exists \varepsilon'$, such that $\Gamma \vdash \varepsilon' <: \varepsilon$, and $\Gamma \mid \Sigma' \vdash e' : \{\varepsilon'\}\ \tau$.

Theorem 5 (Progress)**.** If $\varnothing \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ (i.e., e is a closed, well-typed expression), then either

1. e is a value (i.e., a location) or
2. $\forall \mu$ such that $\mu : \Sigma$, $\exists e', \mu'$ such that $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$.

Chapter 4

Abstract Algebraic Effects via Embeddings

4.1 Background and Motivation

Algebraic effects (introduced by Plotkin and Power [24]) and handlers (introduced by Plotkin and Pretnar [26]) are an approach to computational effects based on a premise that impure behavior arises from a set of operations, and are recently gaining popularity due to their ability to model various form of computational effects such as exceptions, mutable states, async-await, etc.

Modularity is a key concept that separates abstract algebraic effects from the traditional way of using monad to model effects in purely functional programming [28]. However, similar to the restrictive strand of work on effects, few works on algebraic effects have investigated the algebraic effect system on a larger scale, where abstraction between program components is important.

Abstract algebraic effects are first introduced in Biernacki et al. [2]. Similar to abstract types, abstract algebraic effects allow program components to define abstract effect signatures that are opaque to other components in the system. The difference between concrete and abstract effect signatures lies in the ability of program components to handle them. If an effect signature is concrete to a program component, then the operations are accessible to the component, and a handler can handle the effect by handling the operations in the signature. On the other hand, if an effect signature is abstract to one program component, then the component should not observe the operations defined in the effect signature, and is therefore unable to handle the effect. As abstraction is an important issue for module systems because it provides a separation of implementation details of functions from the interface, the abstraction of algebraic effects provides a similar benefit for modularity because it helps separate the component operations from the effect signature, ensuring that the client can only use the handler provided by the library to handle the effect.

The following code is a motivating example similar to the example in [2] that illustrates the challenges of implementing abstract algebraic effects. `Nondet` is a globally defined effect signature. Then we define a module `m` with type `M` with an abstract effect `E`, a method `mflip`, and a handler method `handle`. The effect `E` is defined by `Nondet`, but is opaque to the outside world of the module, because `E` is defined as an abstract effect in the type `M`. The method `mflip` simply calls the `flip` operation, and the `handle` method handles the `flip` operation by returning `true`.

```
effect Nondet {
    flip(): Bool
}

type M
  effect E
  def mflip() : {this.E} Bool
  def handler(Unit -> {this.E} Bool) : {} Bool

module m: M
  effect E = {Nondet}
  def mflip() : {this.E} Unit
    flip()
  def handle(c: Unit -> {this.E} Bool) : {} Bool =
    handle c() with
    | flip() -> resume true

m.handle(
  () => handle m.mflip() with
        | flip() ->  resume false
        | return x -> x
)
```

The last segment of the example shows a client code of module `m` that calls the method `m.handle` and pass in an expression that encapsulates the call to `m.mflip` with another handler that handles the `flip` operation. Since the effect of the method `m.mflip` is abstract, the inner handler should not handle the operation inside `m.mflip`. Instead, the operation should be handled by the outer handler method `m.handle`.

As we can see, the abstraction of effect signatures differs from type abstractions, since the erasure of type information would make the abstraction unsound. So we need a language that keeps track of the information on effect abstraction during the evaluation of the program. In this work, we incorporate the method of syntactic type abstraction introduced by Grossman et al. [10], who use the notion of principals to track the flow of values with abstract types during the evaluation of a program.

4.2 A Simple Example of the Agent-based Language

Consider the simple case where we only have two agents, namely the client `c`, and the host `h`. And the host `h` defines an abstract effect `E`, exports a method that causes the effect, and a method that handles the effect.

```
module h =
  effect E = ...
  val m : 1 -> {E} 1 = ..
  ...
```

Now consider client code wants to handle the `m` function from h

```
handle
  [m ()]^E_h
with
  op (x) -> ...
  return x -> ...
```

Since function `m` is called in the client code, we use a language construct called embedding to encapsulate the function call. The subscript `h` of the embedding indicates that the code inside the embedding is the host code, and the superscript `E` indicates that the effect of the code is `E`, which is abstract to the client. So the embedding ensures that the client would not be able to handle the operation inside the function `m`, therefore keeping the effect abstraction safe.

Besides making the client unable to handle an abstract effect, we need to make sure that a host code can "rediscover" the effect exported by itself. The scenario would be exhibited by the following host code. The client code we showed earlier is now embedded into a host handler that handles the effect E. Because the outer-most handler is now in host code, it would be able to handle the effect operation inside the function `m`.

```
handle
  [handle
    [m ()]^E_h
  with
    op (x) -> ...
    return x -> ...]^E_c
with
  op (x) -> ...
  return x -> ...
```

4.3 Core Calculus

4.3.1 Syntax

This section describes a variant of the simply typed lambda calculus that maintains a syntactic distinction between agents during evaluation. Figure 4.1 gives the syntax of our calculus. As our previous discussion, it is crucial to keep track of the effect abstraction information during the evaluation of the program. It is therefore natural to divide the code into agents and allow each agent to export abstract effect signatures. We assume that there are n agents, and variables i, j, k range over the set of agents.

Every term in this language is assigned to an agent. And terms are split into inert expressions and potentially effectful computations, following an approach called *fine-grain call-by-value*, introduced by Levy et al. [14]. We use the notation *i-expression* and *i-computation* to denote expressions and computations in the agent i. We use subscripts to indicate a term is assigned to an agent, however, we will omit the subscript if the agent the term belongs to is not important or obvious in the context.

An *i-value* is an *i-expression* that cannot be further reduced. There are two forms of *i-value*: the unit $()$, and the lambda abstraction $\lambda x_i : \tau.\ c_i$. *i-expressions* include variable x_i, value v_i, and embedded expressions $[e_j]^\tau_j$. *i-computations* are the terms that can potentially cause

$$
\begin{array}{ll}
\text{(agents)} & i, j ::= \{1 \ldots n\} \\
\text{(lists)} & l ::= i \mid il \\
\text{(value types)} & \tau ::= 1 \mid \tau \to \sigma \\
\text{(computation types)} & \sigma ::= \{\varepsilon\}\tau \\
\text{(effect types)} & \varepsilon ::= \cdot \mid f, \varepsilon \mid op, \varepsilon \\
\text{(i-values)} & v_i ::= ()_i \mid \lambda x_i : \tau.\, c_i \\
\text{(i-expressions)} & e_i ::= x_i \mid v_i \mid [e_j]^{\tau}_{j} \\
\text{(i-computations)} & c_i ::= \texttt{return}\ e_i \mid op(e_i, y.c_i) \mid \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c'_i \mid e_i\ e'_i \\
 & \quad \mid \texttt{with}\ h_i\ \texttt{handle}\ c_i \mid\ [c_j]^{\sigma}_{j} \mid [op]^{\varepsilon}_{l}(e_i, y_i.c_i) \\
\text{(i-handlers)} & h_i ::= \texttt{handler}\ \{\texttt{return}\ x_i \mapsto c^r_i, op^1(x^1_i, k^1) \mapsto c^1_i, \ldots, op^n(x^n_i, k^n) \mapsto c^n_i\}
\end{array}
$$

Figure 4.1: Syntax for multi-agent calculus

effects, and consists of return statement $\texttt{return}\ e_i$, operation call $op(e_i; y_i.c_i)$, sequencing $\texttt{do}\ x_i \leftarrow c_i\ \texttt{in}\ c'_i$, application $e_i\ e'_i$, handling $\texttt{with}\ h_i\ \texttt{handle}\ c_i$, embedded computation $[c_j]^{\sigma}_{j}$, and embedded operation call $[op]^{\varepsilon}_{l}(e_i; y_i.c_i)$. There are a few things worth mentioning:

Sequencing: In $\texttt{do}\ x \leftarrow c\ \texttt{in}\ c'$, we first evaluate c, bind the return value of c' to x and then evluate c_2

Operation Calls: The call $op(e; y.c)$ passes the parameter e to the operation op, binds the return value of the operation call to y, and continue by evaluating the computation c. Note that the encompassing handler could potentially change the behavior of the operation. Explicit continuations greatly simplifies the operational semantics of the language, because continuations make the order of the execution of operations explicit.

Embeddings: the term $[e_j]^{\tau}_{j}$ is an i-expression, where e_j is an embedded j-expression. The type τ is exported by the agent j as the type of the embedded expression. Similarly, $[c_j]^{\sigma}_{j}$ is an embedded j-computation with exported type σ.

Embedded Operations: The embedded operation $[op]^{\varepsilon}_{l}(e; y.c)$ is an operation call that is annotated with effect ε. l is a list of agents that have contributed to the formation of the annotation. The embedded operations should not appear in the source code, as they are an intermediate form of computation that keeps track of the effect annotation of operations. More details of this construct are given in section 4.4 on Operational Semantics.

Similar to terms, types are also divided into expression types and computation types. There are two forms of expression types τ: the unit type 1, and the arrow type $\tau \to \sigma$. As for the computation type σ, there is only one form: $\{\varepsilon\}\tau$, where ε is a set of effects that the computation might induce, and τ is the type of the return value of the computation.

The effect type ε represents an unordered set of effects that can be empty $\cdot$. A effect type can be extended by either an effect label f, or an operation op.

The i-handler h_i must contain a return clause $\texttt{return}\ x \mapsto c$, which handles the case when the handled computation directly returns a value. The returned value is bound to the variable x,

and the entire handling computation evaluates to the computation c. A handler may also contain clauses that handle operations. For example, the clause $op(x, k) \mapsto c$ handles the operation op. More details can be found in the dynamic semantics section.

4.3.2 Agent-Specific Type Information

We use agents to model a module system where each module can have private information about effect abstraction. Each agent in our language has limited knowledge of effect abstraction. For example, an agent i might knows that effect `Nondet = flip(): Bool`, and an agent j does not have this information. As a result, agent i would be able to handle a computation with effect `Nondet`, while the agent j would not be able to handle the effect, because from agent j's point of view, the effect `Nondet` is an abstract label without an operation. Furthermore, we need to ensure the consistency of the information on effect abstraction, that is, agent j should not think that the effect `Nondet = read(): String`, which would contradict with the knowledge of agent i.

The model of effect abstraction information is similar to the model of type information in [10]. To capture effect abstraction information, each agent i has a partial function δ_i that maps an effect label to an effect type. There are two requirements for these maps: (1) For each effect label f, the if there are two agents that knows the implementation of the effect f, then their knowledge about the implementation must be the same. (2) For each effect label f, there is a unique and most concrete interpretation of f. We would not allow the effect label f itself to appear in the implementation of f. Examples like $\delta_i(f) = \{f\}$ and $\delta_i(f) = \{f, op\}$ would be rejected.

Definition 4.3.1. A set $\{\delta_1, \ldots, \delta_n\}$ of maps from effect labels to effects is compatible if

1. For all $i, j \in 1 \ldots n$ if $f \in Dom(\delta_i) \cap Dom(\delta_j)$, then $\delta_i(f) = \delta_i(j)$.
2. Effect labels can be totally ordered such that for every agent i and effect label f, all effect labels in $\delta_i(f)$ precede f.

Then we define the a total function Δ_i that refines an effect type:

Definition 4.3.2. Δ_i is a function that maps an effect type to another effect type, using the effect abstraction knowledge of the agent i.

$$\begin{aligned}
\Delta_i(\cdot) &= \cdot \\
\Delta_i(op, \varepsilon) &= op, \Delta_i(\varepsilon) \\
\Delta_i(f, \varepsilon) &= \begin{cases} f, \Delta_i(\varepsilon) & \text{if } f \notin Dom(\delta_i) \\ \varepsilon', \Delta_i(\varepsilon) & \text{if } \delta_i(f) = \varepsilon' \end{cases}
\end{aligned}$$

The definition of compatibility ensures that there is a fixpoint for repeatedly refining an effect label ε using the function Δ_i. We call such fixpoint $\overline{\Delta_i}(\varepsilon)$.

Definition 4.3.3. $\overline{\Delta_i}(\varepsilon') = \varepsilon$ if there is some $n \geq 0$

$$\underbrace{\Delta_i(\ldots(\Delta_i(\varepsilon')\ldots)}_{\text{n applications}} = \underbrace{\Delta_i(\ldots(\Delta_i(\varepsilon')\ldots)}_{\text{n+1 applications}} = \varepsilon$$

To see how we apply Δ to reach a fix point, consider two effect labels f and g and an agent i. Agent i knows that the effect f is implemented by two operations op_1 and op_2. Then consider

$\boxed{e \longrightarrow e'}$

$$\frac{e_j \mapsto e'_j}{[e_j]^\tau_j \longrightarrow [e'_j]^\tau_j}\ \text{(E-CONGRUENCE)} \qquad \frac{}{[()_j]^1_j \longrightarrow ()_i}\ \text{(E-UNIT)}$$

$$\frac{}{[\lambda x_j : \tau'.\, c_j]^{\tau\to\sigma}_j \longrightarrow \lambda x_i : \tau.\, [\{[x_i]^{\tau'}_i / x_j\} c_j]^\sigma_j}\ \text{(E-LAMBDA)}$$

Figure 4.2: Operational Semantics for Expressions

applying Δ_i to the effect type f, g. We get $\Delta_i(f, g) = op_1, op_2, g$. Since op_1, op_2, and label g is the most concrete form of effects, Δ_i cannot further refine the resulting type, so we have reached the fix point. So we have $\overline{\Delta_i}(f, g) = op_1, op_2, g$. As we can see, by repeatedly applying Δ_i and getting to a fixpoint, we effectively collect all operations and abstract effect labels in the agent i's perspective.

We assume that the type information for operations is public to all agents. The type for an operation op is contained by a separate map Σ, which maps an operation op to an arrow type $\tau_A \to \tau_B$. Note that this is different from the function type in our calculus, which has the form $\tau \to \sigma$.

4.4 Operational Semantics

The reduction rules for terms are dependent on the agent of the terms. Figure 4.2 shows that operational semantics for expressions of agent i. (E-Congruence) shows that a j-expression embedded agent-i should be evaluated using the reduction rules for agent j first. The (E-Unit) and (E-Lambda) rules show that we can lift an embedded j-value to agent i, so the value becomes an i-value. The (E-Unit) rule simply lifts the unit value out of the embedding. The (E-Lambda) rule is more interesting: The value embedded is a lambda expression of agent j. We lift the argument out of the embedding. However, the type annotating the argument is changed from τ' to the exported argument type τ, because the reduced expression should have the exported type $\tau \to \sigma$. The body of the reduced expression is an embedded j-computation, so the variable x_i should be encapsulated by an embedding, because any i-term should be embedded in a j-term. We annotate x_i with type τ' because the original lambda function expects a value of type τ'.

Figure 4.3 shows the reduction rules for i-computations. (E-Ret) is the congruence rule that evaluates the expression in a return statement. (E-Op) evaluates the input argument for the operation call. Note that there is no non-congruence reduction rule for operation calls because the semantics for operations are defined by the handler encapsulating it.

(E-Embed1) is the congruence rule for embedded computations. (E-Embed2) lifts the return statement out of the embedding. We can safely remove the effect annotation ε because the statement that returns a value v_j cannot cause any effect.

(E-Embed3) lifts an operation call out of the embedding. This rule introduces embedded operation as a new language construct. We annotate the embedded operation with the effect annotation of the whole computation. Since the argument value for op is a j-value, we need to embed it as an i-value, and annotate it with type τ_A. The continuation c_j is still embedded, and

$\boxed{c \longrightarrow c'}$

$$\frac{e_i \mapsto e_i'}{\texttt{return}\ e_i \longrightarrow \texttt{return}\ e_i'}\ \text{(E-RET)} \qquad \frac{e_i \mapsto e_i'}{op(e_i, y_i.c_i) \longrightarrow op(e_i', y_i, c_i)}\ \text{(E-OP)}$$

$$\frac{c_j \longrightarrow c_j'}{[c_j]_l^\sigma \longrightarrow [c_j']_l^\sigma}\ \text{(E-EMBED1)} \qquad \frac{}{[\ \texttt{return}\ v_j]_l^{\{\varepsilon\}\tau} \longrightarrow\ \texttt{return}\ [v_j]_l^\tau}\ \text{(E-EMBED2)}$$

$$\frac{\Sigma(op) = \tau_A \to \tau_B}{[op_j(v_j; y_j.c_j)]_j^{\{\varepsilon\}\tau} \longrightarrow [op_j]_j^\varepsilon([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_j^{\{\varepsilon\}\tau})}\ \text{(E-EMBED3)}$$

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Delta_j(\varepsilon') = \varepsilon' \quad op \notin \varepsilon'}{[[op_k]_l^{\varepsilon'}(v_j; y_j.c_j)]_j^{\{\varepsilon\}\tau} \longrightarrow [op_k]_{lj}^\varepsilon([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_j^{\{\varepsilon\}\tau})}\ \text{(E-EMBED4)}$$

$$\frac{e_i \longrightarrow e_i'}{[op]_l^\varepsilon(e_i; y_i.c_i) \longrightarrow [op]_l^\varepsilon(e_i'; y_i.c_i)}\ \text{(E-EMBEDOP1)} \qquad \frac{\overline{\Delta_i}(\varepsilon) = \varepsilon'}{[op]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow [op]_l^{\varepsilon'}(v_i; y_i.c_i)}\ \text{(E-EMBEDOP2)}$$

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \in \varepsilon}{[op]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow op(v_i; y_i.c_i)}\ \text{(E-EMBEDOP3)} \qquad \frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon \quad op' \in \varepsilon}{[op]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow [op]^{\varepsilon \setminus op'}(v_i; y_i.c_i)}\ \text{(E-EMBEDOP4)}$$

$$\frac{e_i \longrightarrow e_i''}{e_i\ e_i' \longrightarrow e_i''\ e_i'}\ \text{(E-APP1)} \qquad \frac{e_i \longrightarrow e_i'}{v_i\ e_i \longrightarrow v_i\ e_i'}\ \text{(E-APP2)} \qquad \frac{}{(\lambda x_i : \tau.\ c_i)\ v_i \longrightarrow \{v_i/x_i\}c_i}\ \text{(E-APP3)}$$

$$\frac{c_i \longrightarrow c_i''}{\texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' \longrightarrow \texttt{do}\ x \leftarrow c_i''\ \texttt{in} c_i'}\ \text{(E-SEQ1)} \qquad \frac{}{\texttt{do}\ x \leftarrow \texttt{return}\ v_i\ \texttt{in}\ c_i' \longrightarrow \{v_i/x\}c_i'}\ \text{(E-SEQ2)}$$

$$\frac{}{\texttt{do}\ x \leftarrow op_i(v_i; y_i.c_i)\texttt{in}\ c_i' \longrightarrow op_i(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')}\ \text{(E-SEQ3)}$$

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{do}\ x \leftarrow [op_j]_l^\varepsilon(v_i; y_i.c_i)\texttt{in}\ c_i' \longrightarrow [op_j]_l^\varepsilon(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')}\ \text{(E-SEQ4)}$$

$$\frac{c_i \longrightarrow c_i'}{\texttt{with}\ h_i\ \texttt{handle}\ c_i \longrightarrow\ \texttt{with}\ h_i\ \texttt{handle}\ c_i'}\ \text{(E-HANDLE1)}$$

$$\frac{\texttt{return}\ x_i \mapsto c_i' \in h_i}{\texttt{with}\ h_i\ \texttt{handle}\ \texttt{return}\ v_i \longrightarrow \{v_i/x_i\}c_i'}\ \text{(E-HANDLE2)}$$

$$\frac{op(x_i; k) \mapsto c_i' \in h_i \quad \Sigma(op) = \tau_A \to \tau_B}{\texttt{with}\ h_i\ \texttt{handle}\ op(v; y_i.c_i) \longrightarrow \{v_i/x_i\}\{(\lambda y_i : \tau_B.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i)/k\}c_i'}\ \text{(E-HANDLE3)}$$

$$\frac{op(x_i; k) \mapsto c_i' \notin h_i}{\texttt{with}\ h_i\ \texttt{handle}\ op(v_i, y_i.c_i) \longrightarrow op(v_i; y_i.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i))}\ \text{(E-HANDLE4)}$$

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{with}\ h_i\ \texttt{handle}\ [op]_l^\varepsilon(v_i, y_i.c_i) \longrightarrow [op]_l^\varepsilon(v_i; y_i.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i))}\ \text{(E-HANDLE5)}$$

Figure 4.3: Operational Semantics for Computations

we substitute the embedded variable y_i for y_j. y_i should be embedded because i-values should be embedded in a j-value.

The (E-Embed4) rule is very similar to (E-Embed3) with one difference being that op is already embedded. In this case, we override the annotation on op with the annotation for the whole computation and update the agent list in the subscript of the operation. We add j to the agent list because the agent j has contributed to the effect annotation of the operation.

(E-EmbedOp1) evaluates the argument of an embedded operation. (E-EmbedOp2) refines the effect annotation of an operation by looking up the effect ε from the type information of the agent i, Δ_i. (E-EmbedOp3) lifts the operation out of an embedding when the annotation contains the operation, because the agent i has enough information about effect abstraction to handle the operation. Note that in the premise, we require that the effect annotation cannot be further refined, in order to ensure determinism of evaluation. (E-EmbedOp4) removes an operation that is not op out of the effect annotation. This step does not affect the correctness of the type information, and is helpful in our proof of type soundness.

(E-App1), (E-App2) and (E-App3) are standard call-by-value semantics for applications. (E-Seq1) evaluates the first computation in a sequence of computations. (E-Seq2) binds the return value of the first computation to a variable in the second computation. (E-Seq3) witnesses an operation call as the first computation in a sequence. Since there is no way to further evaluate an operation right away, we propagate the operation call outwards and defer further evaluation to the continuation of the call. (E-Seq4) is similar to (E-Seq3), and requires that the effect annotation on the embedding to be the most concrete annotation.

(E-Handle1) simply evaluates the computation encapsulated by the handler. In (E-Handle2), the computation returns a value, so we substitute the value into the computation of the clause that handles the return statement in the handler. (E-Handle3) shows that case when the handler h_i has a matching clause for the operation op. We substitute the argument v_i for x_i, and substitute the continuation of the operation for k. The continuation function receives an argument of type τ_B, which is the result type of the operation op, and computes the continuation of the operation c_i, but encapsulates the computation with the handler h_i. The (E-Handle4) shows the case when the operation is not handled by the handler, so we propagate the operation outwards to wait for another handler to handle it. The (E-Handler5) ensures that abstracted effects are not handled: If the current agent cannot refine the effect annotation, then the operation is abstract and cannot be handled, and is therefore propagated outward.

4.5 Static Semantics

4.5.1 Typing Rules

Figure 4.4 shows the static semantics of the core-calculus. Static semantics includes typing rules for both expressions and computations. Note that the typing rules depend on the agent each expression or computation belongs to. We assume that the following rules assign types to terms of agent i.

The rule (T-Unit) assigns the unit type 1 to a unit value. (T-Var) looks up a type of a variable from the context. (T-Lam) is the standard rule for typing a lambda function. Note that the body of

$\boxed{\Gamma \vdash e_i : \tau}$

$$\frac{}{\Gamma \vdash ()_i : 1}\ \text{(T-UNIT)} \quad \frac{}{\Gamma \vdash x_i : \Gamma(x_i)}\ \text{(T-VAR)} \quad \frac{\Gamma, x_i : \tau \vdash c_i : \sigma}{\Gamma \vdash (\lambda x_i : \tau.\ c_i) : \tau \to \sigma}\ \text{(T-LAM)}$$

$$\frac{\Gamma \vdash e_j : \tau' \quad \Gamma \vdash \tau' \leq_{ji} \tau}{\Gamma \vdash [e_j]^{\tau}_{j} : \tau}\ \text{(T-EMBEDEXP)}$$

$\boxed{\Gamma \vdash c_i : \sigma}$

$$\frac{\Gamma \vdash e_i : \tau \quad \Delta_i(\varepsilon) = \varepsilon}{\Gamma \vdash \texttt{return}\ e_i : \{\varepsilon\}\tau}\ \text{(T-RET)} \quad \frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau \quad op \in \Delta_i(\varepsilon)}{\Gamma \vdash op(e_i; y_i.c_i) : \{\varepsilon\}\tau}\ \text{(T-OP)}$$

$$\frac{\Gamma \vdash c_i : \{\varepsilon\}\tau \quad \Gamma, x_i : \tau \vdash c_i' : \{\varepsilon\}\tau'}{\Gamma \vdash \texttt{do}\ x_i \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon\}\tau'}\ \text{(T-SEQ)} \quad \frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2\ : \sigma}\ \text{(T-APP)}$$

$$\frac{\begin{array}{c} h_i = \{\ \texttt{return}\ x \mapsto c^r, op^1(x;k) \mapsto c^1, \ldots, op^n(x;k) \mapsto c^n\} \\ \Gamma, x : \tau_A \vdash c^r : \{\varepsilon'\}\tau_B \quad \left\{\Sigma(op^i) = \tau_i \to \tau_i' \quad \Gamma, x : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c^i : \{\varepsilon'\}\tau_B\right\}_{1 \leq i \leq n} \\ \Gamma \vdash c_i : \{\varepsilon\}\tau_A \quad \varepsilon \setminus \{op^i\}_{1 \leq i \leq n} \subseteq \varepsilon' \end{array}}{\Gamma \vdash \texttt{with}\ h_i\ \texttt{handle}\ c_i : \{\varepsilon'\}\tau_B}\ \text{(T-HANDLE)}$$

$$\frac{\Gamma \vdash c_j : \sigma' \quad \Gamma \vdash \sigma' \leq_{li} \sigma}{\Gamma \vdash [c_j]^{\sigma}_{l} : \sigma}\ \text{(T-EMBED)}$$

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]^{\varepsilon}_{l}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}\ \text{(T-EMBEDOP)}$$

Figure 4.4: Static Semantics

a lambda is computation, so we need to use the typing judgment for computation in the premise of this rule. (T-EmbedExp) assigns type to expression embeddings: The embedding has type τ if the embedded expression e_j is assigned to the type τ', and τ is related to τ' by the list of agents li. We will elaborate on type relations later.

(T-Ret) assigns a type to a return statement: As expected, the expression part of the computation type matches the type of the returned expression. However we can annotate the return statement with an arbitrary effect set, because our type system does not describe the precise effect in computations, but gives the upper bound of effect in computations.

(T-Op) shows the typing rule for operation calls. Again, since we allow effect annotations to be an upper bound on effect, we can require the operation op to be in the effect set ε.

(T-Handle) shows the typing rule for the effect handling statement $\texttt{with}\ h_i\ \texttt{handle}\ c_i$. c_i is a computation with effect type ε and return type τ_A. h_i is a handler that contains a clause that handles operations $op^1, \ldots, op^n$. For the return clause, given the type of variable x is τ_A, the type

of c^r must be $\{\varepsilon'\}\tau_B$. For the clause handling the operation op^i, which has the operation type $\tau_i \to \tau_i'$, if variable x has type τ_i, and continuation has the type $\tau_i' \to \{\varepsilon'\}\tau_B$, then the handling computation c^i must have the type $\{\varepsilon'\}\tau_B$. The effect type after handling, ε' should contain all of the effects that are not handled by the handler.

(T-Embed) is very similar to (T-EmbedExp), where we use the type of the embedded computation and the type relation judgment to assign a type to the embedding.

(T-EmbedOp) first computes the type of c_i given that y_i has the correct type. It is required that the effect annotation on the embedding is a part of the effect of c_i. And op should be related to the effect annotation by the type relations.

4.5.2 Type Relations

$\boxed{\tau \leq_l \tau'}$

$$\frac{}{1 \leq_l 1}\ \text{(R-UNIT)} \qquad \frac{\tau \leq_l \tau' \quad \sigma \leq_l \sigma'}{\tau \to \sigma \leq_l \tau' \to \sigma'}\ \text{(R-ARROW)}$$

$\boxed{\sigma \leq_l \sigma}$

$$\frac{\varepsilon \leq_l \varepsilon' \quad \tau \leq_l \tau'}{\{\varepsilon\}\tau \leq_l \{\varepsilon'\}\tau'}\ \text{(R-SIGMA)}$$

$\boxed{\varepsilon \leq_l \varepsilon}$

$$\frac{\overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon')}{\varepsilon \leq_i \varepsilon'}\ \text{(R-EFF1)} \qquad \frac{\varepsilon \leq_l \varepsilon'' \quad \varepsilon'' \leq_{l'} \varepsilon'}{\varepsilon \leq_{ll'} \varepsilon'}\ \text{(R-EFF2)}$$

Figure 4.5: Type Relations

Type relations ensure the soundness of abstraction. The goal of type relations is to prohibit embeddings from exporting incorrect effect abstractions. For example, if a i-computation uses an effect operation $flip : 1 \to bool$, which is an operation of effect `Nondet`, then whenever it is embedded in another agent, it should be annotated with effect `Nondet`, but should not be annotated with the empty effect or other effects that does not contain `Nondet`.

The judgements for expression types are of the form $\tau \leq_l \tau$, where l is a list of agents that provide the type abstraction information used by the relation. (R-Unit) shows that unit type relates to itself. (R-Arrow) relates two arrow types given that the input types and the output types are related, To relate two computation types, we just need to ensure the effect types and return types are related.

The relation for effect types does the actual work. By (R-EFF1), two effect types are related under a single agent i if the first effect is a subset of the second effect after refinement by the type information provided by i. (R-EFF2) shows that by using type information from a list of agents, we can combine the chain of relation between effects.

4.6 Safety Properties

In this section, we state the standard type-safety theorems for the core calculus and a theorem that shows if an effect is abstract to a client, then the client would never handle the operation of that effect. Since we split terms into expressions and computations, we state progress and preservation lemmas separately. The proofs to theorems stated in this section can be found in appendix C.

4.6.1 Type Soundness

We begin by stating the preservation and progress lemmas. The preservation lemma for expressions and computations are rather standard, except for the fact that we consider terms for each agent separately. In lemma 6, if an i-expression steps to another i-expression, then the new expression should have the identical type as the original expression. Similarly, in lemma 7, the new computation should have the same effect type and return type as the original computation.

Lemma 6 (Preservation for Expressions)**.**
For all agent i, if $\Gamma \vdash e_i : \tau$ and $e_i \mapsto e_i'$, then $\Gamma \vdash e_i' : \tau$.

Lemma 7 (Preservation for Computations)**.**
For all agent i, if $\Gamma \vdash c_i : \{\varepsilon\}\tau$ and $c_i \longrightarrow c_i'$, then $\Gamma \vdash c_i' : \{\varepsilon\}\tau$

The progress lemma for expressions (lemma 8) is also standard. If an i-expression is well-typed, then it either steps to another expression or is already a value.

Lemma 8 (Progress for Expressions)**.**
For agent i, if $\varnothing \vdash e_i : \tau$ then either $e_i = v_i$ or $e_i \longrightarrow e_i'$.

The progress lemma for computations (lemma 9) is a bit more involved. For any well-typed i-computation, there are four possibilities. Similar to expressions, a computation can evaluate to another computation. Otherwise, a computation could potentially be a return statement, an operation, or an embedded operation. The dynamic semantics ensures that these are the only possible final configurations for a computation.

Lemma 9 (Progress for Computations)**.**
If $\varnothing \vdash c_i : \{\varepsilon\}\tau$ then either

1. $c_i \longrightarrow c_i'$
2. $c_i = \texttt{return}\, v_i$
3. $c_i = op(v_i; y_i.c_i')$
4. $c_i = [op]_l^{\varepsilon}(v_i; y_i.c_i')$

4.6.2 Abstraction Safety

In this section, we prove a theorem on the correctness of effect abstraction. The intuition is that if a computation contains operations that are part of an opaque abstract effect, i.e., the definition of the effect is hidden from the agents in the computation, then the evaluation computation would not be affected by the operations.

We first define the notion of *oblivious*. A computation is oblivious to an effect if and only if the effect is not known to the agent i, or any agent that lives inside the computation.

Definition 4.6.1. An i-computation c is oblivious to effect label f if $f \notin Dom(\delta_i)$, and for all subexpression $[e]^{\tau}_j$ and subcomputation $[c]^{\sigma}_j$, $f \notin Dom(\delta_j)$

In the following theorem, we first define an equivalence relation. Two terms are equivalent if they are identical or the effect operations inside the terms are opaque to the agents in the terms. We show that two equivalent terms will stay equivalent after evaluation.

Theorem 10. Let c_1 and c_2 be computations that are oblivious to the effect f. If $c_1 \approx c_2$, $c_1 \to c_1'$, $c_2 \to c_2'$, then $c_1' \approx c_2'$. Furthermore, If e_1, e_2 oblivious to f, $e_1 \approx e_2$, $e_1 \to e_1'$, $e_2 \to e_2'$, then $e_1' \approx e_2'$

The relation $\approx$ is defined as follows:

$\boxed{e \approx e}$

$$\frac{}{x \approx x}\ \text{(R-VAR)} \qquad \frac{}{() \approx ()}\ \text{(R-UNIT)}$$

$$\frac{c \approx c'}{\lambda x : \tau.\, c \approx \lambda x : \tau.\, c'}\ \text{(R-LAM)} \qquad \frac{e \approx e'}{[e]^{\tau}_l \approx [e']^{\tau}_l}\ \text{(R-EMBEDEXP)}$$

$\boxed{c \approx c}$

$$\frac{e \approx e'}{\texttt{return}\, e \approx\ \texttt{return}\, e'}\ \text{(R-RET)} \quad \frac{e \approx e' \quad c \approx c'}{op''(e; y.c) \approx op''(e'; y.c')}\ \text{(R-OP)}$$

$$\frac{c \approx c' \quad d \approx d'}{\texttt{do}\, x \leftarrow c\, \texttt{in}\, d \approx\ \texttt{do}\, x \leftarrow c'\, \texttt{in}\, d'}\ \text{(R-SEQ)} \quad \frac{e_1 \approx e_1' \quad e_2 \approx e_2'}{e_1\ e_2 \approx e_1'\ e_2'}\ \text{(R-APP)}$$

$$\frac{h \approx h' \quad c \approx c'}{\texttt{with}\, h\, \texttt{handle}\, c \approx\ \texttt{with}\, h'\, \texttt{handle}\, c'}\ \text{(R-HANDLE)} \quad \frac{c_j \approx c_j'}{[c_j]^{\sigma}_l \approx [c_j']^{\sigma}_l}\ \text{(R-EMBED)}$$

$$\frac{e \approx e' \quad c \approx c'}{[op'']^{\varepsilon}_l(e; y.c) \approx [op'']^{\varepsilon}_l(e'; y.c')}\ \text{(R-EMBEDOP1)} \quad \frac{e \approx e' \quad c \approx c' \quad \exists i \in l, \delta_i(f) = op, op'}{[op]^{\varepsilon}_l(e; y.c) \approx [op']^{\varepsilon}_l(e'; y.c')}\ \text{(R-EMBEDOP2)}$$

$\boxed{h \approx h}$

$$\frac{c_r \approx c_r' \quad c_1 \approx c_1', \ldots c_n \approx c_n'}{\begin{array}{c}\{\ \texttt{return}\ x \mapsto c_r, op_1(x_1, k_1) \mapsto c_1, \ldots, op_n(x_n, k_n) \mapsto c_n\} \approx \\ \{\ \texttt{return}\ x \mapsto c_r', op_1(x_1, k_1) \mapsto c_1', \ldots, op_n(x_n, k_n) \mapsto c_n'\}\end{array}}\ \text{(R-HANDLER)}$$

Figure 4.6: Definition of equivalence relation $\approx$

Proof. (Sketch) By induction on derivation of $c_1 \approx c_2$ and $e_1 \approx e_2$

1. R-Ret: The only reduction rule that applies is E-Ret, so we have $e_1 \longrightarrow e_1'$ and $e_2 \longrightarrow e_2'$. By IH, we have $e_1' \approx e_2'$. Then the result follows by R-Ret
2. R-Op: The only reduction rule that applies is E-Op. The result is immediate by IH.

3. R-Seq: If the reduction rule is E-Seq1, then result is immediate by IH.
   If the reduction rule is E-Seq2. Then we have $c_1 = \texttt{do}\ x \leftarrow \texttt{return}\ v_1\ \texttt{in}\ d_1$, $c_2 = \texttt{do}\ x \leftarrow \texttt{return}\ v_2\ \texttt{in}\ d_2$. By inversion, we have $v_1 \approx v_2$ and $d_1 \approx d_2$. So we have $\{v_1/x\}d_1 \approx \{v_2/x\}d_2$.
   If the reduction rule is E-Seq3, then $c_1 = \texttt{do}\ x \leftarrow op(v_1; y.k_1)\ \texttt{in}\ d_1$, $c_2 = \texttt{do}\ x \leftarrow op(v_2; y.k_2)\ \texttt{in}\ d_2$. By inversion, we have $k_1 \approx k_2$ and $d_1 \approx d_2$. So $\texttt{do}\ x \leftarrow k_1\ \texttt{in}\ d_1 \approx \texttt{do}\ x \leftarrow k_2\ \texttt{in}\ d_2$. So $op(v_1; y.\ \texttt{do}\ x \leftarrow k_1\ \texttt{in}\ d_1) \approx op(v_2; y.\ \texttt{do}\ x \leftarrow k_2\ \texttt{in}\ d_2)$. The proof is similar for rule E-Seq4.
4. R-App: The cases for reduction rules E-App1 and E-App2 follows by IH. If reduction rule is E-App3. Then $c_1 = (\lambda x : \tau.\ d_1)\ v_1$ and $c_2 = (\lambda x : \tau.\ d_2)\ v_2$. By inversion we have $d_1 \approx d_2$ and $v_1 \approx v_2$. So we have $\{v_1/x\}d_1 \approx \{v_2/x\}d_2$.
5. R-Handle: If reduction rule is E-Handle1, then result follows by IH.
   If the reduction rule is E-Handle2. Then $c_1 = \texttt{with}\ h_1\ \texttt{handle}\ \texttt{return}\ v_1$ and $c_2 = \texttt{with}\ h_2\ \texttt{handle}\ \texttt{return}\ v_2$. By inversion we have $h_1 \approx h_2$, $v_1 \approx v_2$. Let $\texttt{return}\ c_{r1} \in h_1$ and $\texttt{return}\ c_{r2} \in h_2$. By inversion we have $c_{r1} \approx c_{r2}$. So $\{v_1/x\}c_{r1} \approx \{v_2/x\}c_{r2}$.
   If the reduction rule is E-Handle3. Then $c_1 = \texttt{with}\ h_1\ \texttt{handle}\ op(v_1; y_1.c_1)$, and $c_2 = \texttt{with}\ h_2\ \texttt{handle}\ op(v_2; y_2.c_2)$. By inversion we have $h_1 \approx h_2$, $v_1 \approx v_2$, and $c_1 \approx c_2$. Let $op(x_1; k_1) \mapsto c_1' \in h_1$ and $op(x_2; k_2) \mapsto c_2' \in h_2$. By inversion on R-Handler we have $c_1' \approx c_2'$. Then we can conclude that the evaluated computations are still equivalent:
   $$\{v_1/x_1\}\{(\lambda y_1 : \tau_B.\ \texttt{with}\ h_1\ \texttt{handle}\ c_1)/k\}c_1' \approx \{v_2/x_2\}\{(\lambda y_2 : \tau_B.\ \texttt{with}\ h_2\ \texttt{handle}\ c_2)/k\}c_2'$$
   If the reduction rule is E-Handle4. Then $c_1 = \texttt{with}\ h_1\ \texttt{handle}\ op(v_1; y.k_1)$ and $c_2 = \texttt{with}\ h_2\ \texttt{handle}\ op(v_2; y.k_2)$. By inversion we have $v_1 \approx v_2$, $k_1 \approx k_2$ and $h_1 \approx h_2$. Then by equivalence rules we have $c_1' \approx c_2'$. The case for E-Handle5 is similar.
6. R-Embed: If reduction is E-Embed1, result is immediate by IH. If reduction rule is E-Embed2, then $c_1 = [\ \texttt{return}\ v_1]_l^{\{\varepsilon\}\tau}$ and $c_2 = [\ \texttt{return}\ v_2]_l^{\{\varepsilon\}\tau}$. It is easy to see $[v_1]^\tau \approx [v_2]^\tau$. So the result holds.
   If the reduction rule is E-Embed3, Then $c_1 = [op(v_1; y.k_1)]\{\varepsilon\}\tau_l$ and $c_2 = [op(v_2; y.k_2)]_l^{\{\varepsilon\}\tau}$. By inversion we have $v_1 \approx v_1$, $k_1 \approx k_2$. Then by equivalent rules we have $c_1' \approx c_2'$. Same arguments apply for E-Embed4.
7. R-EmbedOp1: If reduction rule is E-EmbedOp1, then result follows by IH. If reduction rules is E-EmbedOp2 or E-EmbedOp3, reduction does not affect terms except effect annotation, so the equivalence relation still hods after reduction.
8. R-EmbedOp2: Reduction rules E-EmbedOp1 and E-EmbedOp2 are similar to the previous case. If the reduction rule is E-EmbedOp3, then by R-EmbedOp2, the operations op and op' are exported as effect f by some agent, and since current agent is oblivious to f, this case is impossible.

□

4.7 Translation of the Abstraction Problem

In this section we show the process of evaluation of the example program presented in 4.1. The original example could be rewritten as follows:

```
op : 1 -> 1

module b: B
  effect f = op
  def m() : {f} Unit
    op ()
  def handler(c: 1 -> {f} 1) : {} Int =
    handle c () with
    | op () -> 1
    | return _ -> 0
```

The operation `op` is defined globally, and the module `b` defines effect `f` to be equivalent to `op`, an effectful method `m`, and an handler method `method`. The example program that we will evaluate is written as follows. The handler method from module `b` is invoked, and the argument is a computation that calls the method `b.m`, which is surrounded by a handler that handles `op`.

```
b.handler(
  () => handle b.m() with
          | op -> resume ()
          | return _ -> ()
 )

```

Then we rewrite the example program in our agent-based calculus: Since the code is a client of `b`, any method call from the module `b` should be surrounded by an embedding. So the handler functions is wrapped in an embedding with type annotation `(1 -> {f} -> 1) -> Int`, and the handled function in the argument is also embedded with an annotation `1 -> {f} 1`. In the following evaluation process we assume that there is an agent `b` that represents the module `b`, and an agent `a` that represents the client code that invokes functions from module `b`

```
// Translation of Example Program
[λc: 1 -> {f} 1.
  handle c() with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{(1→{f}1)→{}int
(λ_:1.
  handle [λ_:1. op()]_b^{1→{f}1} () with
  | op(x, k) -> k ()
  | return _ -> return ())

```

Then according to the dynamic rules, we evaluate the handler function to a value that is not embedded. The program is therefore evaluated to

```
 (λc: 1 -> {f} 1.
  [handle [c]_a^{1→{f}1}  () with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int)
```

```
(λ_:1.
  handle [λ_:1. op()]_b^{1→{f}1} () with
  | op(x, k) -> k ()
  | return _ -> return ())

```

Since the function is evaluated to a value, we can perform a β-reduction:

```
  [handle
     [λ_:1.
      handle [λ_:1. op()]_b^{1→{f}1} () with
      | op(x, k) -> k ()
      | return _ -> return () ]_a^{1→{f}1}  ()
  with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int

```

Now we need to evaluate the outter-most handle computation in the embedding. The first step is to evaluate the handled computation, which is a function application. We first evaluate the function

```
[handle
     λ_:1.
      [handle [λ_:1. op()]_b^{1→{f}1} () with
      | op(x, k) -> k ()
      | return _ -> return () ]_a^{f}1  ()
  with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int

```

Then we pass in the argument, which is a unit value

```
[handle
      [handle [λ_:1. op()]_b^{1→{f}1} () with
      | op(x, k) -> k ()
      | return _ -> return () ]_a^{f}1
  with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int
```

Then we evaluate the inner handling computation. Since the inner handling computation is evaluated as a client code and the operation `op` in an embedding from module `b`, the operation will not be handled by the current handler. Instead, it would be lifted out of the handler.

```
[handle
      [op_b^f((), y. handle return y with
                      | op(x, k) -> k ()
                      | return _ -> return ()
          )]_a^{f}1
  with
```

```
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{{}int
```

At this point since the operation `op` is lifted to the agent b, where the effect f is transparent, the handler would be able to handle it. The result of this computation is `return 1`, because the continuation is discarded by the handler.

```
[handle
    op_ba^f((), y. [handle return y with
                       | op(x, k) -> k ()
                       | return _ -> return ()]_a^{{f}1
        )
with
| op(x, k) -> return 1
| return x -> return 0)]_b^{{}int
```

Chapter 5

Algebraic Effects with Existential Types

5.1 Motivation

In section 3.2.2, we defined a resource type `Var` that provides two public functions, `get` and `set`, that are annotated with effects, but the concrete implementation that causes effects is not revealed to the client. This construction ensures that any program that reads or writes to a value of type `Var` is annotated with the correct effect. Therefore we can see that, in a restrictive effect system, it is useful to be able to hide the implementation of a computational effect.

In this section, we will show that this form of implementation hiding is also important for building modular software with algebraic effects and handlers through an example of mutable state. Mutable state as an algebraic effect is often represented by a `state` effect with two operations `get` and `set`. In this example, we assume that our mutable state can store or access an integer.

```
operation get : Unit -> Int
operation set : Int -> Unit
```

The handler of a state effect is usually defined in the following way:

```
handler hstate = {
  return x -> λ_:Int. return x
  get(_; k) -> λs:Int. (k s) s
  set(s; k) -> λ_:Int. (k ()) s
}
```

This handler would transform the handled computation into a lambda expression that receives a state as an argument. In the return clause, the argument is ignored. In the clause handling `get`, the state argument is passed into the continuation k. Since the continuation $k\ s$ is already transformed into a function that expects a state, we pass s to the computation $k\ s$. The clause for the `set` operation is similar except that we ignore the state argument, but pass the argument obtained from the `set` operation.

Now we consider the module of a single variable introduced in section 3.2.2, where read and write to the variable cannot bypass the effect checking because the implementation details are hidden by the interface `Var`. If we would write a similar module in our core calculus, it is natural to extend our calculus with record type and implement the module as follows:

```
val var =
  < read ↪ λx:Unit. get(); y.return y),
    write ↪ λx:Int. set(x; y.return ()),
    handle ↪  λc: Unit → {state} Int. (with hstate handle c ()) 0 >
```

The module provides functions `read`, `write`, and a handler function `handle` that determines the semantics for operations in functions `read` and `write`. However, this encoding of the module `var` does not enforce that the operations `get` and `set` are always handled by the `handle` function. In fact, any client that calls function `read` and `write` can write its own handler to handle the effects. So in order to solve this issue, we introduce existential type to define the module:

```
val var =
  pack
  <{get, set},
    < read ↪ λx:Unit. get(); y.return y),
      write ↪ λx:Int. set(x; y.return ()),
      handle ↪  λc: Unit → {state} Int. (with hstate handle c ()) 0 >
  > as ∃state. ((read : Unit -> {state} Int)
                ×(write : Int -> {state} Unit)
                ×(handle :  Unit -> {state} Int -> {} Int))
```

This encoding of module defines an abstract effect `state` on top of the effects `get` and `set`, and export the module as an existential type that hides the definition of effect `state`, therefore enforcing that the client of this module can only handle effect `state` by calling the `handle` function.

The embedding design presented in section 4.3 helps to ensure that the abstraction does not break. Imagine a client of module `var` that calls `read` and handles it by calling `handle`.

```
open var as (state, v) in (v.handle v.read)
```

In this example, the `state` effect in function `read` is correctly handled by the handler `handle`, and the result of the computation is 0. Note that the abstraction barrier is still preserved if some handler attempts to handle the abstract effect. For example, let handler `hget` be a handler that handles effect `get`. And the client code uses `hget` to handle the effect in method `read`.

```
handler hget = {
  return x -> x
  get (x; k) -> 1
}

open var as (state, v) in
  (v.handle (λ x: Unit. with hget handle v.read ()))
```

Assume that the open expression is an i-expression. When opening the module `var`, we assign the opened expression `v` to a new agent j that knows the definition of the `state` effect. Because the `hget` handler is evaluating in the agent i, it would not be able to handle the effect `state` of the function `v.read`. Instead, the `state` effect will be handled by the handler provided by the module `var`, therefore ensuring that the abstraction information is not leaked.

```
resource type Logger
  effect ReadLog
  effect UpdateLog
  effect readLog(): {this.ReadLog} String
  effect updateLog(newEntry: String): {this.UpdateLog} Unit

module def logger(f: File): Logger
  effect ReadLog = {f.Read}
  effect UpdateLog = {f.Append}
  def readLog(): {ReadLog} String = f.read()
  def updateLog(newEntry: String): {UpdateLog} Unit = f.append(newEntry)

resource type File
  effect Read
  effect Write
  effect Append
  ...
  def read(): {this.Read} String
  def write(s: String): {this.Write} Unit
  def append(s: String): {this.Append} Unit
  ...
```

Figure 5.1: The logging facility in the text-editor application

5.2 Encoding Abstract Effects Using Algebraic Effects

Our discussion of abstraction of algebraic effects has been focusing exclusively on purely functional programming. However, as shown in Melicher et al. [19], the expressiveness provided by abstract effects enables programmers to develop secure programs when side-effects are in play. In this section, we show that the effect system in this chapter provides a foundation for expressing abstract effect in chapter 3.

Melicher et al. [19] and chapter 3 presented a design of effect member to support effect abstraction: the ability to define higher-level effects in terms of lower-level effects, to hide that definition from clients of an abstraction, and to reveal partial information about an abstract effect through effect bounds. In this chapter we no longer use the object-oriented formalization but use the agent-based lambda calculus introduced in chapter 4 and extend it with the existential types as a foundation to support effect abstraction. Now we will look at different aspects of the original abstract effect system and discuss how we can incorporate them in the new setting with algebraic effects.

5.2.1 Running Example

We begin by encoding the running example presented in Melicher et al. [19] which demonstrates the key feature of abstract effects. The original example shows a type and a module implementing the logging facility in the text-editor application and is shown in figure 5.1.

Consider the code in Fig. 5.1 that shows a type and a module implementing the logging facility of the text-editor application. In the given implementation of the `Logger` type, the `logger`

module accesses the log file. All modules of type `Logger` must have two methods: the `readLog` method that returns the content of the log file and the `updateLog` method that appends new entries to the log file. In addition, the `Logger` type declares two *abstract* effects, `ReadLog` and `UpdateLog`, that are produced by the corresponding methods. These effects are abstract because they are not given a definition in the `Logger` type, and so it is up to the module implementing the `Logger` type to define what they mean. The effect names are user-defined, allowing the choice of meaningful names.

The `logger` module implements the `Logger` type. To access the file system, an object of type `File` (shown in Fig. 2.2) is passed into `logger` as a parameter. The `logger` module's effect declarations are those of the `Logger` type, except now they are *concrete*, i.e., they have specific definitions. The `ReadLog` effect of the `logger` module is defined to be the `Read` effect of the `File` object, and accordingly, the `readLog` method, which produces the `ReadLog` effect, calls `f`'s `read` method. Similarly, the `UpdateLog` effect of the `logger` module is defined to be `f.Append`, and accordingly, the `updateLog` method, which produces the `UpdateLog` effect, calls `f`'s `append` method. In general, effects in a module or object definition must always be concrete, whereas effects in a type definition may be either abstract or concrete.

Using the existential type, the type `Logger` can be translated to the following type, note that we only translate one abstract effect `ReadLog` and one method `readLog` to make the code more readable. We assume that the type String is built into the language.

```
type Logger = ∃ReadLog.⟨readLog : Unit → {ReadLog} String⟩
```

Similarly, the `File` type can be implemented as follows. Again we leave only one abstract effect and one member function to maintain simplicity of the example.

```
type File = ∃Read.⟨read : Unit → {Read} String⟩
```

Then we are finally able to encode the functor `logger`, which receives a value of type `File` and returns a `Logger`.

```
logger : File → {} Logger =
  λf:File. open f as (fRead, fBody) in
  pack (fRead, ⟨readLog ↪ λx: Unit. fBody.read ()⟩) as
    ∃ReadLog.⟨readLog : Unit → {ReadLog} String⟩
```

5.2.2 Effect Abstraction

In section 2.1.3, we defined effect abstraction as the ability to define higher-level effects in terms of lower-level effects and potentially to hide that definition from clients of an abstraction. In the `logger` module above, we lifted the low-level file resource into a higher-level logging facility, and defined higher-level effects `ReadLog` as an abstraction of the lower-level effect `Read`. So application code can reason in terms of effects of higher-level resources when appropriate.

5.2.3 Effect Aggregation

In section 2.1.4 we argued that effect aggregation can make the effect system less verbose. The original example declares an effect `UpdateLog` as the sum of two effects `f.Read` and `f.Write`.

```
module def logger(f: File): Logger
effect UpdateLog = {f.Read, f.Write}
def updateLog(newEntry: String): {this.UpdateLog} Unit
...
```

Although our new calculus is inherently more verbose than the language based on path-dependent effects, it can still encode effect aggregation. First we let the `File` type contains two abstract effects by existential quantification:

```
type File = ∃Read. ∃Write. ...
```

Then we can define logger functor. We first open the module `f`, then define an existential package where the abstract effect is defined as a sum of the two effects from the module `f`:

```
logger : File → {} Logger =
  λ f : File. open f as (fRead, x) in open x as (fWrite, y) in
  pack ({fRead, fWrite}, ⟨updateLog↪ ...⟩) as
    (∃UpdateLog. ⟨updateLog: String → {UpdateLog} Unit⟩)
```

Again the functor `logger` receives a `File` and returns a `Logger`. The functor opens the `f` module twice to get the two effect labels `fRead` and `fWrite` that are exported by `f`. Then the `logger` returns an existential package that hides the two effect as an abstract effect `UpdateLog`. This design achieves effect aggregation by combining the two lower-level effects into one higher-level effect.

5.3 Formalization

In the previous sections, we have introduced the core calculus of abstract algebraic effects via embeddings. However it is impractical for requiring programmers to explicitly annotate each program component with embeddings. So we present a top level language where the annotations are implicitly added during the evaluation of the program.

According to Mitchell and Plotkin [21], there is a correspondence between abstract data types and existential types. Existential types are often used as a foundation for expressing type abstraction in module systems. The calculus introduced in this section contains a form of existential type that provides abstraction mechanisms for algebraic effects. The values that have existential type would generate new agents during the evaluation of the program and automatically separate program components with different knowledge on effect abstraction, so programmers would not need to explicitly work with agents and embeddings.

5.3.1 Syntax

Most of the syntax remains the same for our new language. The existential type $\exists f.\ \tau$ is added as an expression type. The intuition of the type is that the value of this type is a value of type $\{\varepsilon/f\}\tau$ for some effect type ε.

There are two new forms of expressions: The introduction form of the existential type, `pack` (ε, e) `as` $\exists f.\ \tau$ and the elimination form, `open` e `as` (f, x) `in` e'. The `pack` expression creates existential package that contains an effect type ε and an expression e. The `open`

$$
\begin{array}{lll}
\text{(agents)} & & i, j ::= \{1 \ldots n\} \\
\text{(lists)} & & l ::= i \mid il \\
\text{(expression types)} & & \tau ::= 1 \mid \tau \to \sigma \mid \exists f.\, \tau \\
\text{(computation types)} & & \sigma ::= \{\varepsilon\}\tau \\
\text{(effect types)} & & \varepsilon ::= \cdot \mid f, \varepsilon \mid op, \varepsilon \\
\text{(i-values)} & & v_i ::= ()_i \mid \lambda x_i : \tau.\, c_i \mid \texttt{pack}\ (\varepsilon, v)\ \texttt{as}\ \exists f.\, \tau \\
\text{(i-expressions)} & & e_i ::= x_i \mid v_i \mid [e_j]^{\tau}_{l} \mid \texttt{pack}\ (\varepsilon, e)\ \texttt{as}\ \exists f.\, \tau \mid \texttt{open}\ e\ \texttt{as}\ (f, x)\ \texttt{in}\ e' \\
\text{(i-computations)} & & c_i ::= \texttt{return}\ e_i \mid op(e_i, y.c_i) \mid \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' \mid e_i\ e_i' \\
& & \quad \mid \texttt{with}\ h_i\ \texttt{handle}\ c_i \mid [c_j]^{\sigma}_{l} \mid [op]^{\varepsilon}_{l}(e_i, y_i.c_i) \\
\text{(i-handler)} & & h_i ::= \texttt{handler}\ \{\texttt{return}\ x_i \mapsto c_i^r, op^1(x_i^1, k^1) \mapsto c_i^1 \ldots op^n(x_i^n, k^n) \mapsto c_i^n\}
\end{array}
$$

Figure 5.2: Syntax for Existential Effects

$\boxed{\langle \{\Delta\}, e\rangle \mapsto \langle \{\Delta\}, e\rangle}$

$$\frac{\langle \{\Delta\}, e\rangle \mapsto \langle \{\Delta'\}, e'\rangle}{\langle \{\Delta\},\ \texttt{pack}\ (\varepsilon, e)\ \texttt{as}\ \exists f.\, \tau\rangle \mapsto \langle \{\Delta'\},\ \texttt{pack}\ (\varepsilon, e')\ \texttt{as}\ \exists f.\, \tau\rangle}\ \text{(E-PACK)}$$

$$\frac{f\ \texttt{fresh} \quad j\ \texttt{fresh} \quad \Delta'_j = \Delta_i[f \mapsto \varepsilon] \quad \forall \Delta_k \in \{\Delta\}, \Delta'_k = \Delta_k[f \mapsto f]}{\langle \{\Delta\},\ \texttt{open}\ (\ \texttt{pack}\ (\varepsilon, e)\ \texttt{as}\ \exists f.\, \tau)\ \texttt{as}\ (f, x)\ \texttt{in}\ e'\rangle \mapsto \langle \{\Delta'\}, \{[e]^{\tau}_{j}/x\}e'\rangle}\ \text{(E-OPEN1)}$$

$$\frac{\langle \{\Delta\}, e\rangle \mapsto \langle \{\Delta'\}, e'\rangle}{\langle \{\Delta\},\ \texttt{open}\ e\ \texttt{as}\ (f, x)\ \texttt{in}\ e''\rangle \mapsto \langle \{\Delta'\},\ \texttt{open}\ e'\ \texttt{as}\ (f, x) \in e''\rangle}\ \text{(E-OPEN2)}$$

$$\frac{}{\langle \{\Delta\}, [\ \texttt{pack}\ (\varepsilon, v)\ \texttt{as}\ \exists f.\, \tau]^{\exists f.\, \tau'}_{j}\rangle \mapsto \langle \{\Delta\},\ \texttt{pack}\ (\varepsilon, [v]^{\{\varepsilon/f\}\tau'}_{j})\ \texttt{as}\ \exists f.\, \tau'\rangle}\ \text{(E-EMBEDPACK)}$$

Figure 5.3: Additional Dynamic Semantics for Existential Type

expression opens up an existential package and substitutes the expression in the package for the variable x.

We only introduce one form of value: the existential package $\texttt{pack}\ (\varepsilon, v)\ \texttt{as}\ \exists f.\, \tau$, where the packed expression is already evaluated to a value.

5.3.2 Dynamic Semantics

The semantics for existential type $\exists f.\, \tau$ hides the definition of the effect label f from the client of the value of this type. We leverage our previous design of multi-agent calculus to achieve information hiding. However, the previous design assumes that the type information for each agents is predetermined and does not change during evaluation. Since existential types generate new abstraction boundaries, we need different semantics that allow agents to be created during

$\boxed{\{\Delta\} \mid \Gamma \vdash e_i : \tau}$

$$\frac{\{\Delta\} \mid \Gamma \vdash e : \{\varepsilon/f\}\tau}{\{\Delta\} \mid \Gamma \vdash \texttt{ pack } (\varepsilon, e) \texttt{ as } \exists f.\, \tau : \exists f.\, \tau} \text{ (T-PACK)}$$

$$\frac{\{\Delta\} \mid \Gamma \vdash e : \exists f.\, \tau \quad \forall \Delta_i \in \{\Delta\}, \Delta_i' = \Delta_i[f \mapsto f] \quad \{\Delta'\} \mid \Gamma, x : \tau \vdash e' : \tau'}{\{\Delta\} \mid \Gamma \vdash \texttt{ open } e \texttt{ as } (f, x) \texttt{ in } e' : \tau'} \text{ (T-OPEN)}$$

Figure 5.4: Additional Static Semantics for Existential Effects

evaluation. Therefore, we modify the reduction rule to evaluate a pair that contains both the expression to evaluate and a context of type information. The idea to keep track of type information while evaluating terms was used by Grossman et al. [10] to encode parametric polymorphism in their system.

In figure 5.3, we show the dynamics semantics for new constructs such as exists and open. In the rules we use the syntax $\Delta_i[f \mapsto \varepsilon]$ to express extending the type map Δ_i with a new projection from label $\texttt{f}$ to effect ε.

We introduce the notation $\{\Delta\}$ to express a list of type maps for all agents in the context $\{\Delta_1, \ldots \Delta_n\}$. The type information of each agent can change, and new agents can be generated, the evaluation judgment now has the form $\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta\}, e\rangle$.

(E-Pack) shows the congruence rule for reduction of a pack expression. (E-Open) opens an existential package: This rule requires f to be a fresh label, which achievable by applying alpha conversion in τ. j is a fresh agent. A new type map for agent j extends the type map for i by mapping f to ε. Every existing type map in $\{\Delta\}$ is extended by mapping f to itself. Finally, $[e]^{\tau}_{j}$ is substituted for x in e'.

(E-EmbedPack) shows how the embedding interacts with existential packages. The existential package is lifted out of the embedding, and the value v becomes an embedded j-value with type annotation $\{\varepsilon/f\}\tau'$.

The reduction rules for remaining terms are not changed by the introduction of $\{\Delta\}$ and are therefore not shown.

5.3.3 Static Semantics

The typing rules also require the set of type maps, so the judgments have the form $\{\Delta\} \mid \Gamma \vdash e : \tau$ (T-Pack) assigns the type $\exists f.\, \tau$ to the existential package if the expression e has type $\{\varepsilon/f\}\tau$. The rule (T-Open) assigns the type τ' to the open expression if e has the existential type $\exists f.\, \tau$ and e' has type τ' given that the context is extended with variable x, and the set of type maps is extended with the effect label f.

Similar to the previous system, type relations ensure the correctness of the type annotation on embeddings. Since we introduced the existential type, we need to add an additional rule to the type relations:

$$\frac{\tau \leq_l \tau'}{\exists f.\, \tau \leq_l \exists f.\, \tau'} \; (R - Exists)$$

5.3.4 Type Safety

Lemma 11. (Preservation for expression)
If $\{\Delta\} \mid \Gamma \vdash e : \tau$ and $\langle \{\Delta\}, e\rangle \mapsto \langle \{\Delta'\}, e'\rangle$, then $\{\Delta'\} \mid \Gamma \vdash e' : \tau$

Proof. By rule induction on the dynamic semantics of expressions.

1. (E-Congruence): By inversion on typing judgement and applying IH.
2. (E-Unit): By directly applying (T-Unit)
3. (E-Lambda):

$$\frac{}{[\lambda x_j : \tau'.\ c_j]_j^{\tau\to\sigma} \longrightarrow \lambda x_i : \tau.\ [\{[x_i]_i^{\tau'}/x_j\}c_j]_{jl}^{\sigma}} \text{ (E-LAMBDA)}$$

   By inversion on (T-EmbedExp), we have $\{\Delta\} \mid \Gamma \vdash [\lambda x_j : \tau'.\ c_j]_j^{\tau\to\sigma} : \tau \to \sigma$, and $\{\Delta\} \mid \Gamma \vdash \lambda x_j : \tau'.\ c_j : \tau' \to \sigma'$, where $\{\Delta\} \mid \Gamma \vdash \tau' \to \sigma' \leq_j \tau \to \sigma$. By inversion on (T-Lam), we have $\{\Delta\} \mid \Gamma, x_j : \tau' \vdash c_j : \sigma$. Then by substitution lemma, we have $\{\Delta\} \mid \Gamma, x_i : \tau \vdash \{[x_i]_i^{\tau'}/x_j\}c_j : \sigma'$. By (T-Embed), $\{\Delta\} \mid \Gamma, x_i : \tau \vdash [\{[x_i]_i^{\tau'}/x_j\}c_j : \sigma']_j^{\sigma} : \sigma$. Then the result follows by (T-Lam).
4. (E-Pack): By inversion and IH.
5. (E-Open): By inversion on (T-Pack), we have $\{\Delta\} \mid \Gamma \vdash e : \{\varepsilon/f\}\tau$. Then by (T-EmbedExp), we have $\{\Delta'\} \mid \Gamma \vdash [e]_j^{\tau} : \tau$. By inversion on (T-Open), $\{\Delta\} \mid \Gamma, x : \tau \vdash e' : \tau'$. Since j is fresh, e' doesn't contain any j term, so the type information from agent j doesn't affect the typing of e'. Therefore, $\{\Delta'\} \mid \Gamma, x : \tau \vdash e' : \tau'$. Finally, by substitution lemma, we have $\{\Delta'\} \mid \Gamma \vdash \{[e]_j^{\tau}/x\}e' : \tau'$.
6. (E-EmbedPack): By inversion on (T-EmbedExp), we have $\{\Delta\} \mid \Gamma \vdash \texttt{pack}\ (\varepsilon, v)\ \texttt{as}\ \exists f.\ \tau : \exists f.\ \tau$, and $\{\Delta\} \mid \Gamma \vdash \exists f.\ \tau \leq_j \exists f.\ \tau'$. By type relation, we have $\tau \leq_j \tau'$. Then by inversion on (T-Pack), we have $\{\Delta\} \mid \Gamma \vdash v : \{\varepsilon/f\}\tau$. Then by T-EmbedExp, we have $\{\Delta\} \mid \Gamma \vdash [v]_j^{\{\varepsilon/f\}\tau'} : \{\varepsilon/f\}\tau'$. Then by (T-Pack), we get $\{\Delta\} \mid \Gamma \vdash \texttt{pack}\ (\varepsilon, [v]_j^{\{\varepsilon/f\}\tau'})\ \texttt{as}\ \exists f.\ \tau' : \exists f.\ \tau'$

□

Lemma 12. (Progress)
If $\{\Delta\} \mid \Gamma \vdash e : \tau$ then either e is a value or there exists e' and $\{\Delta'\}$ such that $\langle \{\Delta\}, e\rangle \mapsto \langle \{\Delta'\}, e'\rangle$

Proof. By induction on the typing rule:

(T-Pack) Let $e = \texttt{pack}\ (\varepsilon, e')\ \texttt{as}\ \exists f.\tau$ There are two cases. If e' is value, then we are done. If e' is not a value, by inversion and IH, we have $\langle \{\Delta\}, e'\rangle \mapsto \langle \{\Delta'\}, e''\rangle$. Then we can apply E-Pack to evaluate e.

(T-Open) Again, let $e = \texttt{open}\ e'\ \texttt{as}\ (f, x) \in e''$. By IH and inversion, if e' is not a value, then we can evaluate e by (E-Open2). If e' is a value, by inversion, $e' = \texttt{pack}\ (\varepsilon, e_1)\ \texttt{as}\ \exists f.\tau$. Then we can evaluate e by applying (E-Open1).

□

5.4 Discussion and Future Work

5.4.1 Parametric Polymorphism

We introduced polymorphic effects in chapter 3 as a part of our restrictive effect system. However, we did not include polymorphism in our agent-based core calculus. Parametric polymorphism on effects would significantly increase the expressiveness of the language. For example, in the example of the `var` module presented in the previous section, the type of the argument to the `handle` function is `Unit -> {state} Int`. So it restricts the handled computation to only have the `state` effect, and is therefore unrealistic as we may want to handle computations with other effects as well.

```
1 val var =
2   pack
3   <{get, set},
4     < read ↪ λx:Unit. get(); y.return y),
5       write ↪ λx:Int. set(x; y.return ()),
6       handle ↪  λc: Unit → {state} Int. (with hstate handle c ()) 0 >
7   > as ∃state. (read : Unit -> {state} Int) * (write : Int -> {state} Unit)
     * (handle :  Unit -> {state} Int -> {} Int)
```

Therefore it would be desirable to add universal quantifications on effect variables to the system. However, unlike existential quantification, which is a straightforward extension to our calculus, the universal effect introduces a problem that breaks the abstraction barrier. The following example illustrates the problem brought by parametric polymorphism on effects:

```
1  type B
2    effect E {
3      def op() : Unit
4    }
5
6  module b : B
7    ...
8
9  type A
10   effect E
11   def handle[F](c: Unit -> {F, this.E} Unit): {F} Unit
12   def m(): {this.E} Unit
13
14 module a: A
15   effect E = {b.E}
16   def handle[F](c: Unit -> {F, this.E} Unit): {F} Unit
17      handle
18        c()
19      with
20        b.op() -> resume ()
21   def m(): {this.E} Unit
22      b.op()
23
```

The module `a` defines a polymorphic handle function that handles a computation with effect `a.E` and a polymorphic effect `F`. Since effect `a.E` is defined abstract in the type `A`, the client of this

module should not observe the fact that effect `a.E` is equivalent to `b.E`. However, the `client1` in the following code passes effect `b.E` as the polymorphic effect into the `handle` function, and because the implementation of `handle` function handles the effect `b.E`, the operation `b.op` would be handled by `a.handle`. So client1 be would be surprised by that the effect `b.E` is handled, and the desired output is not printed. In comparison, the client2 code passes an unrelated effect `c.E` to the handling function and observes the line "desired output" is printed. This example illustrates that the clients can actually observe the implementation of the effect `a.E`, which is supposed to be opaque.

```
1  //Client1: Prints nothing
2  handle
3    a.handle[b.E] (
4       () => b.op(); a.m()
5    )
6  with
7      b.op() -> print "desired output"
8
9  //Client2: Prints "desired output"
10 handle
11   a.handle[c.E] (
12      () => c.op(); a.m()
13   )
14 with
15     c.op() -> print "desired output"
```

5.4.2 Effect Bounds

As we have shown in chapter 3, effect bounds are a useful tool for making the effect system more expressive. Currently, our calculus does not support bounded quantification because its formalization is different from the path-dependent formalization we previously developed to express effect bounds.

One possible direction to achieve this is through the use of bounded existential types. Because we use existential types to express information hiding on effect types, it is natural to adopt the technique of bounded existential types to achieve the idea of effect bounds. It could be interesting to explore how subeffecting introduced by effect bounds interacts with the mechanism of agent-based type information.

Chapter 6

Related Work and Conclusion

6.1 Related Work

6.1.1 Restrictive Effect Systems

A restrictive effect system considers effects that are built into the language, such as reading and writing states or exceptions, and provides a way to restrict the usage of such effects. Our effect system introduced in chapter 3 is a restrictive effect system. Restrictive effect systems were first proposed by Lucassen [16] to track reads and writes to memory. Then Lucassen and Gifford [17] extended this effect system to support polymorphism. Restrictive Effects have since been used for a wide variety of purposes, including exceptions in Java [11] and asynchronous event handling [5].

6.1.2 Bounded Effect Polymorphism.

A limited form of bounded effect polymorphism was explored by Trifonov and Shao [29], who bound effect parameters by the resources they may act on; however, the bound cannot be another arbitrary effect, as in our system. Long et al. [15] use a form of bounded effect polymorphism internally but do not expose it to users of their system.

6.1.3 Subeffecting.

Some effect systems, such as Koka [12], provide a built-in set of effects with fixed sub-effecting relationships between them. Rytz et al. [27] support more flexibility via an extensible framework for effects. Users can plug in their own domain of effects, specifying an effect lattice representing sueffecting relationships. Each plugin is monolithic. In contrast, our effect members allow new effects to be incrementally added and related to existing effects using declared subeffect bounds.

6.1.4 Path-dependent Effects

The Effekt library by Brachthauser et al. [4] explores algebraic effects as a library of the Scala programming language. Since their effect system is built on the path-dependent type system of

Scala, it bears some similarities to our system. However, their work is largely orthogonal to our contributions due to the following reasons:

Our goal is to check the effects of general-purpose code. In contrast, Brachthäuser's approach requires all effect-checked code to be written in a monad. This is required to support control effects (i.e. prescriptive effects), but it is not an incidental difference: it is also an integral part of their static effect checking system, because monads are the way that they couple Scala's type members (which provide abstraction and polymorphism) to effects. Their paper, therefore, does not solve the problem of soundly checking effects for non-monadic code. Most code in Scala and Wyvern–let alone more conventional languages such as Java–is non-monadic, for good reasons: monads are restrictive and, for some kinds of programming, quite awkward. Programmers may be willing to use monads narrowly to get the benefits of control effects that Brachthäuser et al. support, but outside the Haskell community, it does not seem likely they would be willing to use them at a much broader scale for the purpose of descriptive effects.

Our approach provides abstraction and polymorphism for descriptive effects. As discussed above, Brachthäuser et al.'s leverage of Scala's type members provides abstraction and polymorphism only for prescriptive effects, and only in the context of monadic code. Even setting aside the issue of monads, above, it is unclear how their approach can provide abstraction for descriptive effects. The reason is that their abstraction works backwards from the kind we need. For example, we want to be able to implement a logger in terms of file I/O–and hide the fact that it is implemented that way. The natural way to start would be to model file I/O in their system as a set of effect operations that "handle" I/O operations at the top level (their system does not provide support for this, so it would have to be added). A logger library could then provide a set of "log" effects and a handler for them, implemented in terms of the top-level I/O operations. But it would not be possible to hide the fact that the logger library was implemented in terms of the I/O operations, because the handler for the log effects would have to be annotated with I/O operation effects. Furthermore, all log operations would have to be nested in the scope of the log handler, annoyingly inverting control flow relative to the expected approach. And this would have to be done for every library that abstracts from the built-in I/O effects, a highly anti-modular approach.

6.1.5 Abstract Algebraic Effects

Our discussion in chapter 4 tackles the issue of abstraction of algebraic effects. This issue was originally raised by Leijen [13], but was not discussed in depth. Biernacki et al. [2] introduced a core calculus called λ^{HEL} with abstract algebraic effects. However, there are multiple distinctions that distinguish our core calculus from λ^{HEL}. First, we adopted the agent-based syntax that syntactically distinguishes each module by assigning them to different agents. This design allows us to reason about the code using the information provided by agents, and provide syntactic proof for abstraction properties. Moreover, our calculus can be simply extended with existential types, which serves as an abstraction to represent module systems for a high-level language. The benefit of this design is that the programmer would not need to write embeddings explicitly, as embeddings are generated as a semantic object when a value of existential type is evaluated. In comparison, it is unclear from the paper [2] how a top-level language with a module system could be translated to the coercion-based calculus λ^{HEL}.

Zhang and Myers [32] describe a design of algebraic effects that preserves abstraction in the setting of parametric functions. If a function does not statically know about an algebraic effect, that effect tunnels through that function. This is different from our form of abstraction, in which the definition of an effect is hidden from clients.

6.2 Conclusion

Effect systems have been actively studied for nearly four decades, but they are not widely used in the software development process because little attention is paid to improve the usability of effect systems when developing large and complex software. On the other hand, type abstraction is an invaluable tool to software designers, which enables programmers to reason about interfaces between different components of programs. Therefore, we explored different ways to achieve effect type abstraction within existing frameworks of computational effects.

This thesis presented the Bounded Abstract Effects, an effect system that allows effects to be defined as members of objects. The types for objects in our language serves as the interface that can hide the implementation of an effect. Effects are declared as members of an object type, and effect members can be declared abstractly with upper and lower bounds. Our system enables effect abstraction between different program components and allows building a hierarchy of abstract effects via effect bounds.

We have also presented a core calculus for abstract algebraic effects, which ensures the correctness of abstraction by using embeddings to keep track of the type information during evaluation. We have provided a syntactic proof of the correctness of the abstraction barrier and have shown that a portion of our design of Bounded Abstract Effects can be implemented within the setting of algebraic effects and handlers. Moreover, we have added the existential types for effects to our system as a foundation for abstract effect types.

Bibliography

[1] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 233–249, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660216. URL https://doi.org/10.1145/2660193.2660216. 1, 2.1.2

[2] Dariusz Biernacki, Maciej Piròg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting Algebraic Effects. In *Symposium on Principles of Programming Languages*, 2019. 1, 2.2.2, 4.1, 6.1.5

[3] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Object Oriented Programming Systems Languages and Applications*, 2009. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640097. URL http://doi.acm.org/10.1145/1640089.1640097. 1

[4] JONATHAN IMMANUEL Brachthauser, PHILIPP SCHUSTER, and KLAUS OSTERMANN. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. *Journal of Functional Programming*, 30:e8, 2020. doi: 10.1017/S0956796820000027. 6.1.4

[5] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile Event Correlation with Algebraic Effects. *Proceedings of the ACM on Programming Languages*, 2(ICFP):67:1–67:31, 2018. ISSN 2475-1421. doi: 10.1145/3236762. URL http://doi.acm.org/10.1145/3236762. 1, 2.1, 6.1.1

[6] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming*, 2017. doi: 10.1007/978-3-319-89719-6_6. URL https://doi.org/10.1007/978-3-319-89719-6_6. 1

[7] Andrzej Filinski. Monads in Action. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 483–494, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605584799. doi: 10.1145/1706299.1706354. URL https://doi.org/10.1145/1706299.1706354. 1

[8] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. Javaui: Effects for

controlling ui object access. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, pages 179–204, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39038-8. 3.2.1

[9] Aaron Greenhouse and John Boyland. An object-oriented effects system. pages 205–229, 06 1999. doi: 10.1007/3-540-48743-3_10. 3.2.2

[10] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst.*, 22(6):1037–1080, November 2000. ISSN 0164-0925. doi: 10.1145/371880.371887. URL https://doi.org/10.1145/371880.371887. 1, 4.1, 4.3.2, 5.3.2

[11] Joseph R. Kiniry. *Advanced Topics in Exception Handling Techniques*, chapter Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. Springer-Verlag, 2006. ISBN 3-540-37443-4, 978-3-540-37443-5. URL http://dl.acm.org/citation.cfm?id=2124243.2124264. 2.1, 6.1.1

[12] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. In *Mathematically Structured Functional Programming*, 2014. URL https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/. 6.1.3

[13] Daan Leijen. Algebraic effect handlers with resources and deep finalization. Technical Report MSR-TR-2018-10, April 2018. URL https://www.microsoft.com/en-us/research/publication/algebraic-effect-handlers-resources-deep-finalization/. 2.2.2, 6.1.5

[14] Paul Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185, 08 2003. doi: 10.1016/S0890-5401(03)00088-9. 4.3.1

[15] Yuheng Long, Yu David Liu, and Hridesh Rajan. Intensional effect polymorphism. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 346–370. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.346. URL https://doi.org/10.4230/LIPIcs.ECOOP.2015.346. 6.1.2

[16] John M. Lucassen. *Types and Effects towards the Integration of Functional and Imperative Programming*. PhD thesis, Massachusetts Institute of Technology, 1987. 2.1, 6.1.1

[17] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Symposium on Principles of Programming Languages*, 1988. ISBN 0-89791-252-7. doi: 10.1145/73560.73564. URL http://doi.acm.org/10.1145/73560.73564. 1, 2.1, 6.1.1

[18] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A Capability-Based Module System for Authority Control. In *European Conference on Object-Oriented Programming*, 2017. 3.3, 3.3.2, 3.3.6

[19] Darya Melicher, Anlun Xu, Jonathan Aldrich, Alex Potanin, and Zhao Valerie. Bounded

abstract effects: Applications to security. 2020. 1, 2.1, 5.2, 5.2.1

[20] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988. doi: 10.1145/44501.45065. URL https://doi.org/10.1145/44501.45065. 1

[21] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988. ISSN 0164-0925. doi: 10.1145/44501.45065. URL https://doi.org/10.1145/44501.45065. 5.3

[22] E. Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989. doi: 10.1109/LICS.1989.39155. 1

[23] Martin Odersky and Matthias Zenger. Scalable component abstractions. *SIGPLAN Not.*, 40(10):41–57, October 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094815. URL https://doi.org/10.1145/1103845.1094815. 1

[24] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:69–94, 02 2003. doi: 10.1023/A:1023064908962. 4.1

[25] Gordon Plotkin and John Power. Algebraic Operations and Generic Effects. *Applied Categorical Structures*, 11(1):69–94, 2003. ISSN 1572-9095. doi: 10.1023/A:1023064908962. URL https://doi.org/10.1023/A:1023064908962. 2.2.1

[26] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *Programming Languages and Systems*, 2009. ISBN 978-3-642-00590-9. 1, 2.2.1, 4.1

[27] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight Polymorphic Effects. In *European Conference on Object-Oriented Programming*, 2012. ISBN 978-3-642-31056-0. doi: 10.1007/978-3-642-31057-7_13. URL http://dx.doi.org/10.1007/978-3-642-31057-7_13. 6.1.3

[28] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: What binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 98–113, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342595. URL https://doi.org/10.1145/3331545.3342595. 4.1

[29] Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In *European Symposium on Programming Languages and Systems*, 1999. 6.1.2

[30] Franklyn A. Turbak and David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008. ISBN 0262201755, 9780262201759. 1, 2.1

[31] Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. *SIGPLAN Not.*, 40(10):455–471, October 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094847. URL https://doi.org/10.1145/1103845.1094847. 1

[32] Yizhou Zhang and Andrew C. Myers. Abstraction-safe Effect Handlers via Tunneling. *Proceedings of the ACM on Programming Languages*, 3(POPL):5:1–5:29, 2019. ISSN 2475-

1421. doi: 10.1145/3290318. URL http://doi.acm.org/10.1145/3290318.6.1.5

Appendix A

Transitivity of Subtyping

A.1 Lemmas

Lemma 13. If $\Gamma, x : \tau \vdash \varepsilon_1 <: \varepsilon_2$, and $\Gamma \vdash \tau' <: \tau$, then $\Gamma, x : \tau' \vdash \varepsilon_1 <: \varepsilon_2$.

Proof. The proof is by structural induction on the rule to derive $\Gamma, x : \tau \vdash \varepsilon_1 <: \varepsilon_2$.

1. Subeffect-Subset
   Since the premise doesn't rely on the context. This case is trivially true.
2. Subeffect-Upperbound
   If the type of n is not changed, then we can apply the same rule to to derive $\Gamma, x : \tau' \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2$. If the type of n is replaced by τ', then we have $\texttt{effect}\ \ g \leqslant \varepsilon' \in \sigma$, where $\Gamma, n : \tau' \vdash \varepsilon' <: \varepsilon$. By IH, we have $\Gamma, n : \tau' \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2$. By transitivity of subeffecting, we have $\Gamma, n : \tau' \vdash [n/y]\varepsilon' \cup \varepsilon_1 <: \varepsilon_2$. Then we can apply Subeffect-Upperbound again to derive $\Gamma, x : \tau' \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2$.
3. Subeffect-Lowerbound
   This case is similar to Subeffect-Upperbound
4. Subeffect-Def-1
   Since the declaration type $\texttt{effect}\ g = \{\varepsilon\}$ is not changed, the result follows directly by induction hypothesis.
5. Subeffect-Def-2
   Since the declaration type $\texttt{effect}\ g = \{\varepsilon\}$ is not changed, the result follows directly by induction hypothesis.

□

Lemma 14. If $\Gamma, x : \tau \vdash \tau_1 <: \tau_2$, and $\Gamma \vdash \tau' <: \tau$, then $\Gamma, x : \tau' \vdash \tau_1 <: \tau_2$
If $\Gamma, x : \tau \vdash \sigma_1 <: \sigma_2$, and $\Gamma \vdash \tau' <: \tau$, then $\Gamma, x : \tau' \vdash \sigma_1 <: \sigma_1$

Proof. We induct on the number of S-Alg used to derive the typing judgment in the premise of the statement.

BC S-Alg is not used, so we have $\Gamma, x : \tau \vdash \sigma_1 <: \sigma_2$ derived by S-Refl2 or one of the S-Effect rules. The proof is trivial if we apply lemma 13.

IS1 Assume we used S-Alg n times to derive $\Gamma, x : \tau \vdash \{y \Rightarrow \sigma_i^{i\in 1...m}\} <: \Gamma \vdash \{y \Rightarrow {\sigma'}_i^{i\in 1...n}\}$. Then for each subtyping judgments in the premise of S-Alg, we can apply induction hypothesis to derive $\Gamma,\ x : \tau',\ y : \{y \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma_{p(i)} <: \sigma'_i$. Then by applying S-Alg, we have $\Gamma, x : \tau' \vdash \{y \Rightarrow \sigma_i^{i\in 1...m}\} <: \Gamma \vdash \{y \Rightarrow {\sigma'}_i^{i\in 1...n}\}$

IS2 Assume we used S-Alg n times to derive $\Gamma, y : \tau \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 <: \texttt{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2$, by inversion on S-Def, we have $\Gamma, y : \tau \vdash \tau'_1 <: \tau_1$, $\Gamma, y : \tau \vdash \tau_2 <: \tau'_2$, and $\Gamma, y : \tau, x : \tau_1 \vdash \varepsilon_1 <: \varepsilon_2$. Then by induction hypothesis and lemma 13, we have $\Gamma, y : \tau' \vdash \tau'_1 <: \tau_1$, $\Gamma, y : \tau' \vdash \tau_2 <: \tau'_2$, and $\Gamma, y : \tau', x : \tau_1 \vdash \varepsilon_1 <: \varepsilon_2$. Then we use S-Def to derive $\Gamma, y : \tau' \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 <: \texttt{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2$

□

A.2 Proof of Theorem 3

If $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, then $\Gamma \vdash \tau_1 <: \tau_3$.
If $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$, then $\Gamma \vdash \sigma_1 <: \sigma_3$.

Proof. We induct on the the number of S-Alg used to derive the two judgments in the premise of the first statement: $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, or the two judgments in the premise of the second statement: $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$.

BC The S-Alg is not used, so we have $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$ by S-Refl2 or one of S-Effect. By lemma 18 transitivity of subeffecting, it is easy to see $\Gamma \vdash \sigma_1 <: \sigma_3$

IS1 Assume we used S-Alg n times to derive $\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1...m}\} <: \{x \Rightarrow {\sigma'}_i^{i\in 1...n}\}$ and $\Gamma \vdash \{x \Rightarrow {\sigma'}_i^{i\in 1...n}\} <: \{x \Rightarrow {\sigma''}_i^{i\in 1...k}\}$. By inversion of S-Alg, there is an injection $p : \{1..n\} \mapsto \{1..m\}$ such that $\forall i \in 1..n,\ \Gamma, x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma_{p(i)} <: \sigma'_i$. There is another injection $q : \{1..k\} \mapsto \{1..n\}$ such that $\forall i \in 1..k,\ \Gamma, x : \{x \Rightarrow {\sigma'}_i^{i\in 1..n}\} \vdash \sigma'_{q(i)} <: \sigma''_i$. So for each $i \in 1..k$ we have two judgments

$$\Gamma, x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma_{p(q(i))} <: \sigma'_{q(i)}$$
$$\Gamma, x : \{x \Rightarrow {\sigma'}_i^{i\in 1..n}\} \vdash \sigma'_{q(i)} <: \sigma''_i$$

By lemma 14, we can write the second judgment as $\Gamma, x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma'_{q(i)} <: \sigma''_i$. By IH, for all $i \in 1..k$, $\Gamma, x : \{x \Rightarrow {\sigma''}_i^{i\in 1..k}\} \vdash \sigma_{p(q(i))} <: \sigma''_i$. Since the function $p \circ q$ is a bijection from $\{1..k\} \mapsto \{1..n\}$, we can use the rule S-Alg again to derive $\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1...m}\} <: \{x \Rightarrow {\sigma''}_i^{i\in 1...k}\}$

IS2 Assume we used S-Alg n times to derive $\Gamma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau'_1 <: \texttt{def}\ m(x : \tau_2) : \{\varepsilon_2\}\ \tau'_2$ and $\Gamma \vdash \texttt{def}\ m(x : \tau_2) : \{\varepsilon_2\}\ \tau'_2 <: \texttt{def}\ m(x : \tau_3) : \{\varepsilon_3\}\ \tau'_3$. By inverse on S-Def, we have $\Gamma \vdash \tau_2 <: \tau_1$, $\Gamma \vdash \tau_3 <: \tau_2$, $\Gamma \vdash \tau'_1 <: \tau'_2$, and $\Gamma \vdash \tau'_2 <: \tau'_3$. By IH, we have $\Gamma \vdash \tau'_1 <: \tau'_3$ and $\Gamma \vdash \tau_3 <: \tau_1$. We have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$ by transitivity of subeffects. Hence we can use S-Def again to derive $\Gamma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau'_1 <: \texttt{def}\ m(x : \tau_3) : \{\varepsilon_3\}\ \tau'_3$.

IS3 By transitivity of subeffecting, other cases for $\Gamma \vdash \sigma_1 <: \sigma_3$ are trivial.

□

Appendix B

Proofs of the Type Soundness Theorems for Bounded Abstract Effects

B.1 Lemmas

Proof. Straightforward induction on typing derivations. □

Lemma 15 (Weakening)**.** If $\Gamma \mid \varnothing \vdash e : \{\varepsilon\}\ \tau$ and $x \notin dom(\Gamma)$, then $\Gamma,\ x : \tau' \mid \varnothing \vdash e : \{\varepsilon\}\ \tau$, and the latter derivation has the same depth as the former.

Proof. Straightforward induction on typing derivations. □

Lemma 16 (Reverse of SUBEFFECTING-LOWERBOUND)**.** If $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$, $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, and $\texttt{effect}\ \ g \leqslant \varepsilon \in \sigma$ then $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$

Proof. We prove this by induction on $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\})$, which is defined in Fig. 3.6

BC If $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\}) = 0$. Then $x.g$ can not have a definition. This case is vacuously true.

IS We case on the rule used to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$:

(a) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Subset: If $x.g \notin \varepsilon_1$, then we can use Subeffect-Subset to show $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$ If $x.g \in \varepsilon_1$. Then $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$, where $\varepsilon_1' \subseteq \varepsilon_2$. So we can use Subeffect-Def-1 to show $\Gamma \vdash \varepsilon_1' \cup \{x.g\} <: \varepsilon_2 \cup [x/y]\varepsilon$

(b) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Upperbound:
Then we have $\varepsilon_1 = \varepsilon_1' \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ \ h = \{\varepsilon'\} \in \sigma$, and $\Gamma \vdash \varepsilon_1' \cup [z/y']\varepsilon' <: \varepsilon_2 \cup \{x.g\}$ By IH, we have $\Gamma \vdash \varepsilon_1' \cup [z/y']\varepsilon' <: \varepsilon_2 \cup [x/y]\varepsilon$ Using Subeffect-Upperbound, we have $\Gamma \vdash \varepsilon_1' \cup \{z.h\} <: \varepsilon_2 \cup [x/y]\varepsilon$

(c) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Def-1:
If Subeffect-Def-1 uses the effect $x.g$, then we immediately have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$ Otherwise, if Subeffect-Def-1 doesn't use $x.g$, then we have $\varepsilon_2 = \varepsilon_2' \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ \ h = \{\varepsilon'\} \in \sigma$, and $\Gamma \vdash \varepsilon_1 <: \varepsilon_2' \cup [z/y']\varepsilon' \cup \{x.y\}$. By IH, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2' \cup [z/y']\varepsilon' \cup [x/y]\varepsilon$. Using Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$

(d) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Def-2:
This case is similar to (b)

□

Lemma 17 (Reverse of SUBEFFECTING-DEF-2)**.** If $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$, $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, and $\texttt{effect}\ \ g = \{\varepsilon\} \in \sigma$ then $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$

Proof. We prove this by induction on $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\})$, which is defined in Fig. 3.6

BC If $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\}) = 0$. Then $x.g$ can not have a definition. This case is vacuously true.

IS We case on the rule used to derive $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$:

(a) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Subset:
Then $x.g \in \varepsilon_2$. So we can use Subeffect-Def-1 to derive $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$

(b) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Upperbound:
If the Subeffect-Upperbound rule uses the effect $x.g$, then we by the premise of Subeffect-Upperbound, we have $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$ If the Subeffect-Upperbound rule does not use the effect $x.g$, then we have $\varepsilon_1 = \varepsilon_1' \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ \ h \leqslant \varepsilon' \in \sigma$, and $\Gamma \vdash \varepsilon_1' \cup [z/y']\varepsilon' \cup \{x.g\} <: \varepsilon_2$ By IH, we have $\Gamma \vdash \varepsilon_1' \cup [z/y']\varepsilon' \cup [x/y]\varepsilon <: \varepsilon_2$. Using Subeffect-Upperbound, we derive $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$.

(c) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Def-1:
Then we have $\varepsilon_2 = \varepsilon_2' \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ \ h = \{\varepsilon'\} \in \sigma$, and $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2' \cup [z/y']\varepsilon'$. By IH, we have $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2' \cup [z/y']\varepsilon'$. Using Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2 \cup \{z.h\}$.

(d) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Def-2:
This case is similar to (b)

□

Lemma 18 (Transitivity in subeffecting)**.** If $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$, then $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

Proof. We prove this using structural induction on $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3)$, which is defined in Fig. 3.6

BC Let $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3) = 0$. The judgments $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ are derived from Subeffect-Subset. So we have transitivity immediately.

IS Let $N \geq 0$, assume $\forall \varepsilon_1, \varepsilon_2, \varepsilon_3$ with $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3) \leq N$, if $\varepsilon_1 <: \varepsilon_2$ and $\varepsilon_2 <: \varepsilon_3$, then $\varepsilon_1 <: \varepsilon_3$. Let $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ and $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3) = N + 1$. Want to show $\varepsilon_1 <: \varepsilon_3$. We case on the rules used to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$

(a) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Subset

i. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Subset.
Transitivity in this case is trivial.

ii. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Upperbound.
Let $\varepsilon_2 = \varepsilon_2' \cup \{x.g\}$. By Subeffect-Upperbound, we have $\Gamma \vdash x : \{y \Rightarrow \sigma\}$ $\texttt{effect}\ \ g \leqslant \varepsilon \in \sigma$ and $\varepsilon_2' \cup [x/y]\varepsilon <: \varepsilon_3$ There are two cases:

A. If $\{x.g\} \notin \varepsilon_1$, then $\varepsilon_1 \subseteq \varepsilon_2'$. Therefore $\Gamma \vdash \varepsilon_1 <: \varepsilon_2' \cup [x/y]\varepsilon$. By induction hypothesis, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

B. If $\{x.g\} \in \varepsilon_1$, then $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$, and $\varepsilon_1' \subseteq \varepsilon_2'$. So we have $\Gamma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_2' \cup [x/y]\varepsilon$ by Subeffect-Subset. By IH, we have $\varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_3$. Then we use Subeffect-Upperbound to derive $\varepsilon_1' \cup \{x.g\} <: \varepsilon_3$

iii. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-1.
Let $\varepsilon_3 = \varepsilon_3' \cup \{x.g\}$. We have $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, $\texttt{effect}\ \ g = \{\varepsilon\}$, and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3' \cup [x/y]\varepsilon$. By IH, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3' \cup [x/y]\varepsilon$ Then we can use Subeffect-Def-1 again to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$

iv. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-2.
The proof is identical to ii.

(b) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Upperbound.
So we have $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$. $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, $\texttt{effect}\ \ g = \{\varepsilon\}$, and $\Gamma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_2$. Using IH, we have $\Gamma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_3$. Using Suveffect-Upperbound again, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

(c) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Def-1.
Therefore we let $\varepsilon_2 = \varepsilon_2' \cup \{x.g\}$, $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, and $effect\ g = \{\varepsilon\} \in \sigma$. By premise of Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 <: [x/y]\varepsilon \cup \varepsilon_2'$. Since $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$, we have $\Gamma \vdash \varepsilon_2' \cup \{x.g\} <: \varepsilon_3$.

i. $\Gamma \vdash \varepsilon_2' \cup \{x.g\} <: \varepsilon_3$ by Subeffect-Subset
Then we have $\varepsilon_3 = \varepsilon_3' \cup \{x.g\}$, and $\varepsilon_2' \subseteq \varepsilon_3'$. Therefore we have $\varepsilon_2' \cup [x/y]\varepsilon \subseteq \varepsilon_3' \cup [x/y]\varepsilon$. Therefore, $\Gamma \vdash \varepsilon_2' \cup [x/y]\varepsilon <: \varepsilon_3' \cup [x/y]\varepsilon$. By IH, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3' \cup [x/y]\varepsilon$. By Subeffect-Def-1 ,we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3' \cup \{x.g\} = \varepsilon_3$

ii. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Upperbound
Since the effect $\{x.g\}$ is used by Subeffect-Def-1, it is not used by the rule Subeffect-Upperbound. Let $\varepsilon_2 = \varepsilon_2'' \cup \{x.g\} \cup \{z.h\}$. By Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2'' \cup [x/y]\varepsilon \cup \{z.h\}$. By Subeffect-Upperbound, we have $\Gamma \vdash z : \{y' \Rightarrow \sigma'\}$, $\texttt{effect}\ \ h \leqslant \varepsilon' \in \sigma'$, and $\Gamma \vdash \varepsilon_2'' \cup \{x.g\} \cup [z/y']\varepsilon' <: \varepsilon_3$. By Lemma 16 and $\Gamma \vdash \varepsilon_1 <: \varepsilon_2'' \cup [x/y]\varepsilon \cup \{z.h\}$, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2'' \cup [x.y]\varepsilon \cup [z/y']\varepsilon'$. By Lemma 17 and $\Gamma \vdash \varepsilon_2'' \cup \{x.g\} \cup [z/y']\varepsilon' <: \varepsilon_3$, we have $\Gamma \vdash \varepsilon_2'' \cup [x/y]\varepsilon \cup [z/y']\varepsilon' <: \varepsilon_3$. Therefore, we use IH to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

iii. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-1
Therefore, let $\varepsilon_3 = \varepsilon_3' \cup \{z.h\}$, $\Gamma \vdash z : \{y \Rightarrow \sigma'\}$, and $effect\ h = \{\varepsilon'\} \in \sigma'$. And we have $\Gamma \vdash \varepsilon_2 <: \varepsilon_3' \cup \{z.h\}$. By premise of Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_2 <: [z/y]\varepsilon' \cup \varepsilon_3'$. By IH, we have $\Gamma \vdash \varepsilon_1 <: [z/y]\varepsilon' \cup \varepsilon_3'$. Using Subeffect-Def-1, we derive that $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

iv. $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-2
This case is identical to c (ii)

(d) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Def-2
This case is identical to (b)

(e) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Lowerbound
This case is identical to (c)

□

Lemma 19 (Substitution in types)**.** If $\Gamma,\ z : \tau \vdash \tau_1 <: \tau_2$ and $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau$, then $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_2$. Furthermore, if $\Gamma,\ z : \tau \vdash \sigma_1 <: \sigma_2$ and $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau$, then $\Gamma \vdash [l/z]\sigma_1 <: [l/z]\sigma_2$.

Proof. The proof is by simultaneous induction on a derivation of $\Gamma,\ z : \tau \vdash \tau_1 <: \tau_2$ and $\Gamma,\ z : \tau \vdash \sigma_1 <: \sigma_2$. For a given derivation, we proceed by cases on the final typing rule used in the derivation:

Case S-REFL1*:* $\tau_1 = \tau_2$, and the desired result is immediate.

Case S-TRANS*:* By inversion on S-TRANS, we get $\Gamma,\ z : \tau \vdash \tau_1 <: \tau_2$ and $\Gamma,\ z : \tau \vdash \tau_2 <: \tau_3$. By the induction hypothesis, $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_2$ and $\Gamma \vdash [l/z]\tau_2 <: [l/z]\tau_3$. Then, by S-TRANS, $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_3$.

Case S-PERM*:* $\tau_1 = \{x \Rightarrow \sigma_i^{i\in 1..n}\}$ and $\tau_2 = \{x \Rightarrow \sigma_i'^{i\in 1..n}\}$. Substitution preserves the permutation relations, and thus, $[l/z]\{x \Rightarrow \sigma_i^{i\in 1..n}\}$ is a permutation of $[l/z]\{x \Rightarrow \sigma_i'^{i\in 1..n}\}$. Then, by S-PERM, $\Gamma \vdash [l/z]\{x \Rightarrow \sigma_i^{i\in 1..n}\} <: [l/z]\{x \Rightarrow \sigma_i'^{i\in 1..n}\}$.

Case S-WIDTH*:* $\tau_1 = \{x \Rightarrow \sigma_i^{i\in 1..n+k}\}$ and $\tau_2 = \{x \Rightarrow \sigma_i^{i\in 1..n}\}$, and the desired result is immediate.

Case S-DEPTH*:* $\tau_1 = \{x \Rightarrow \sigma_i^{i\in 1..n}\}$ and $\tau_2 = \{x \Rightarrow \sigma'^{i\in 1..n}_i\}$. By inversion on S-DEPTH, we get $\forall i,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..n}\},\ z : \tau \vdash \sigma_i <: \sigma_i'$. By the induction hypothesis, $\forall i,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..n}\} \vdash [l/z]\sigma_i <: [l/z]\sigma_i'$. Then, by S-DEPTH, $\Gamma \vdash [l/z]\{x \Rightarrow \sigma_i^{i\in 1..n}\} <: [l/z]\{x \Rightarrow \sigma_i'^{i\in 1..n}\}$.

Case S-REFL2*:* $\sigma_1 = \sigma_2$, and the desired result is immediate.

Case S-DEF*:* $\sigma_1 = \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2$ and $\sigma_2 = \texttt{def}\ m(x : \tau_1') : \{\varepsilon_2\}\ \tau_2'$. By inversion on S-DEF, we get $\Gamma,\ z : \tau \vdash \tau_1' <: \tau_1, \Gamma,\ z : \tau \vdash \tau_2 <: \tau_2', \Gamma, z : \tau \vdash \varepsilon_1 <: \varepsilon_2$. By the induction hypothesis, $\Gamma \vdash [l/z]\tau_1' <: [l/z]\tau_1$ and $\Gamma \vdash [l/z]\tau_2 <: [l/z]\tau_2'$. By lemma 20, $\Gamma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$. Then, by S-DEF, $\Gamma \vdash [l/z](\texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2) <: [l/z](\texttt{def}\ m(x : \tau_1') : \{\varepsilon_2\}\ \tau_2')$.

Case S-EFFECT*:* $\sigma_1 = \texttt{effect}\ g = \{\varepsilon\}$ and $\sigma_2 = \texttt{effect}\ g$, and the desired result is immediate.

Thus, substituting terms in types preserves the subtyping relationship. □

Lemma 20 (Substitution in expressions and effects)**.** If $\Gamma,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ and $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$, then $\Gamma \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon\}\ [l/z]\tau$.

And if $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \mid \Sigma \vdash l : \{\}[l/z]\tau$, then $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$.

And if $\Gamma,\ z : \tau' \mid \Sigma \vdash d : \sigma$ and $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$, then $\Gamma \mid \Sigma \vdash [l/z]d : [l/z]\sigma$.

Furthermore, if $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon\ wf$, then $\Gamma \mid \Sigma \vdash [l/z]\varepsilon\ wf$

Proof. The proof is by simultaneous induction on a derivation of $\Gamma,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, $\Gamma,\ z : \tau' \mid \Sigma \vdash d : \sigma$, $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_1$, and $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon wf$. For a given derivation, we proceed by cases on the final typing rule used in the derivation:

Case T-VAR*:* $e = x$, and by inversion on T-VAR, we get $x : \tau \in (\Gamma,\ z : \tau')$. There are two subcases to consider, depending on whether x is z or another variable. If $x = z$, then $[l/z]x = l$ and $\tau = \tau'$. The required result is then $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$, which is among the assumptions of the lemma. Otherwise, $[l/z]x = x$, and the desired result is immediate.

Case T-NEW*:* $e = \texttt{new}(x \Rightarrow \overline{d})$, and by inversion on T-NEW, we get $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\},\ z : \tau' \mid \Sigma \vdash d_i : \sigma_i$. By the induction hypothesis, $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \mid \Sigma \vdash [l/z]d_i : [l/z]\sigma_i$. Then, by T-NEW, $\Gamma \mid \Sigma \vdash \texttt{new}(x \Rightarrow [l/z]\overline{d}) : \{\}\ \{x \Rightarrow [l/z]\overline{\sigma}\}$, i.e., $\Gamma \mid \Sigma \vdash [l/z](\texttt{new}(x \Rightarrow \overline{d})) : \{\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$.

Case T-METHOD*:* $e = e_1.m(e_2)$, and by inversion on T-METHOD, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$; $\texttt{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$; $\Gamma,\ z : \tau' \mid \Sigma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf$; and $\Gamma,\ z : \tau' \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$, $\texttt{def}\ m(y : [l/z]\tau_2) : \{[l/z]\varepsilon_3\}\ [l/z]\tau_1 \in [l/z]\overline{\sigma}$, $\Gamma \mid \Sigma \vdash [l/z]([e_1/x][e_2/y]\varepsilon_3)\ wf$, and $\Gamma \mid \Sigma \vdash [l/z]e_2 : \{[l/z]\varepsilon_2\}\ [l/z][e_1/x]\tau_2$. Then, by T-METHOD, $\Gamma \mid \Sigma \vdash [l/z]e_1.m([l/z]e_2) : \{[l/z]\varepsilon_1 \cup [l/z]\varepsilon_2 \cup [l/z]([e_1/x][e_2/y]\varepsilon_3)\}\ [l/z]([e_1/x][e_2/y]\tau_1)$, i.e., $\Gamma \mid \Sigma \vdash [l/z](e_1.m(e_2)) : \{[l/z](\varepsilon_1 \cup \varepsilon_2 \cup [e_1/x][e_2/y]\varepsilon_3)\}\ [l/z]([e_1/x][e_2/y]\tau_1)$.

Case T-FIELD*:* $e = e_1.f$, and by inversion on T-FIELD, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}$ and $\texttt{var}\ f : \tau \in \overline{\sigma}$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$ and $\texttt{var}\ f : [l/z]\tau \in [l/z]\overline{\sigma}$. Then, by T-FIELD, $\Gamma \mid \Sigma \vdash ([l/z]e_1).f : \{[l/z]\varepsilon\}\ [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](e_1.f) : \{[l/z]\varepsilon\}\ [l/z]\tau$.

Case T-ASSIGN*:* $e = (e_1.f = e_2)$, and by inversion on T-ASSIGN, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$; $\texttt{var}\ f : \tau \in \overline{\sigma}$; and $\Gamma,\ z : \tau' \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ \tau$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$; $\texttt{var}\ f : [l/z]\tau \in [l/z]\overline{\sigma}$; and $\Gamma \mid \Sigma \vdash [l/z]e_2 : \{[l/z]\varepsilon_2\}\ [l/z]\tau$. Then, by T-ASSIGN, $\Gamma \mid \Sigma \vdash [l/z]e_1.f = [l/z]e_2 : \{[l/z]\varepsilon_1 \cup [l/z]\varepsilon_2\}\ [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](e_1.f = e_2) : \{[l/z](\varepsilon_1 \cup \varepsilon_2)\}\ [l/z]\tau$.

Case T-LOC*:* $e = l$, $[l/z]l = l$, and the desired result is immediate.

Case T-SUB*:* $e = e_1$, and by inversion on T-Sub, we get $\Gamma, z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\tau_1$, $\Gamma, z : \tau' \mid \Sigma \vdash \tau_1 <: \tau_2$ and $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_2$. By induction hypothesis, we have $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}[l/z]\tau_1$, $\Gamma \mid \Sigma \vdash [l/z]\tau_1 <: [l/z]\tau_2$, and $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$. Then, by T-sub, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_2\}[l/z]\tau_2$

Case DT-DEF*:* By inversion, we have $\Gamma, z : \tau,\ x : \tau_1 \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$, $\Gamma, z : \tau,\ x : \tau_1 \mid \Sigma \vdash \varepsilon\ wf$, $\Gamma, z : \tau \mid \Sigma \vdash \varepsilon' <: \varepsilon$, By IH, we have $\Gamma,\ x : [l/z]\tau_1 \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon'\}\ [l/z]\tau_2$, $\Gamma,\ x : [l/z]\tau_1 \mid \Sigma \vdash [l/z]\varepsilon\ wf$, $\Gamma \mid \Sigma \vdash [l/z]\varepsilon' <: [l/z]\varepsilon$. By DT-Def, we have $\Gamma \mid \Sigma \vdash \texttt{def}\ m(x : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2 = [l/z]e\ :\ \texttt{def}\ m(x : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2$
Case DT-VAR*:* $d = \texttt{var}\ f : \tau = n$, and by definition of n, there are two subcases:

Subcase n is x: In this case, $d = \texttt{var}\ f : \tau = x$, and by inversion on DT-VAR, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash x : \{\}\ \tau$. There are two subcases to consider, depending on whether x is z or another variable. If $x = z$, then by the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]x : \{\}\ [l/z]\tau$, which yields $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau$ and $\tau = \tau'$, and thus, $\Gamma \mid \Sigma \vdash \texttt{var}\ f : [l/z]\tau = l\ :\ \texttt{var}\ f : [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](\texttt{var}\ f : \tau = l)\ :\ [l/z](\texttt{var}\ f : \tau)$, as required. If $x \neq z$, then $\Gamma \mid \Sigma \vdash [l/z]x : \{\}\ [l/z]\tau$ yields $\Gamma \mid \Sigma \vdash x : \{\}\ [l/z]\tau$, and thus, $\Gamma \mid \Sigma \vdash \texttt{var}\ f : [l/z]\tau = x\ :\ \texttt{var}\ f : [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](\texttt{var}\ f : \tau = x)\ :\ [l/z](\texttt{var}\ f : \tau)$, as required.

Subcase n is l: In this case, $d = \texttt{var}\ f : \tau = l$, i.e., the field is resolved to a location l. This is not affected by the substitution, and the desired result is immediate.

Case DT-EFFECT*:* By IH, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon\ wf$. We use DT-Effect to derive $\Gamma \mid \Sigma \vdash \texttt{effect}\ g = \{[l/z]\varepsilon\} : \texttt{effect}\ g = \{[l/z]\varepsilon\}$
Case SUBEFFECT-SUBSET*:* By inversion, we have $\varepsilon_1 \subseteq \varepsilon_2$. So $[l/z]\varepsilon_1 \subseteq [l/z]\varepsilon_2$. By Subeffect-Subset, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$.
Case SUBEFFECT-UPPERBOUND*:* By inversion, we have $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$, $\Gamma, z : \tau \mid \Sigma \vdash x : \{y \Rightarrow \sigma\}$, $effect\ g \leqslant \{\varepsilon\} \in \sigma$ and $\Gamma, z : \tau \mid \Sigma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_2$. By IH, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [l/z][x/y]\varepsilon <: [l/z]\varepsilon_2$. Since y is a free variable, we select y such that $x \neq y$ and $y \neq z$. We case on if $z = x$:

1. If $z \neq x$, then we can swap the order of the substitutions on ε $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [x/y][l/z]\varepsilon <: [l/z]\varepsilon_2$. Using substitution lemma for typing on $\Gamma, z : \tau \mid \Sigma \vdash x : \{y \Rightarrow \sigma\}$, we have $\Gamma \mid \Sigma \vdash x : \{y \Rightarrow [l/z]\sigma\}$, $\texttt{effect}\ g \leqslant [l/z]\varepsilon \in [l/z]\sigma$. Using Subeffect-Upperbound, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup \{x.g\} <: [l/z]\varepsilon_2$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$.
2. If $z = x$ Then we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [l/x, y]\varepsilon <: [l/z]\varepsilon_2$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [l/y][l/z]\varepsilon <: [l/z]\varepsilon_2$ We case on the derivation of $\Gamma, z : \tau \mid \Sigma \vdash z : \{y \Rightarrow \sigma\}$.
   (a) (T-Var)

   $$\frac{z : \tau \in \Gamma, z : \tau}{\Gamma, z : \tau \mid \Sigma \vdash z : \tau}$$

   So $\tau = \{y \Rightarrow \sigma\}$. By our assumption, we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$. Since $\texttt{effect}\ g \leqslant \varepsilon \in \sigma$, we have $\texttt{effect}\ g \leqslant [l/z]\varepsilon \in [l/z]\sigma$. Therefore, we can use Subeffect-Upperbound on $\{l.g\}$ to derive $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup \{l.g\} <: [l/z]\varepsilon_2$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$
   (b) (T-Sub)

$$\frac{\Gamma, z:\tau \mid \Sigma \vdash z:\tau_1 \quad \Gamma, z:\tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}}{\Gamma, z:\tau \mid \Sigma \vdash z:\{y \Rightarrow \sigma\}}$$

Notice that we introduced a new type τ_1 that z can be ascribed to. The judgment $\Gamma, z:\tau \mid \Sigma \vdash z:\tau_1$ can be derived by T-Sub, which introduce a new type τ_2 such that $\Gamma, z:\tau \mid \Sigma \vdash \tau_2 <: \tau_1$, or T-Var, which shows $\tau_1 = \tau$. Therefore if we follow the derivation tree, we get a chain relation $\Gamma, z:\tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}$, $\Gamma, z:\tau \mid \Sigma \vdash \tau_2 <: \tau_1, \ldots, \Gamma, z:\tau \mid \Sigma \vdash \tau <: \tau_n$. We can apply IH on these judgments, so we have a chain $\Gamma \mid \Sigma \vdash [l/z]\tau_1 <: \{y \Rightarrow [l/z]\sigma\}$, $\Gamma \mid \Sigma \vdash [l/z]\tau_2 <: [l/z]\tau_1 \ldots, \Gamma \mid \Sigma \vdash [l/z]\tau <: [l/z]\tau_n$. By transitivity of subtyping, we have $\Gamma \mid \Sigma \vdash [l/z]\tau <: \{y \Rightarrow [l/z]\sigma\}$ So we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$ The rest of the proof is similar to case (a).

Case SUBEFFECT-DEF-1*:* By inversion, we have $\varepsilon_2 = \varepsilon_2' \cup \{x.g\}$, $\Gamma, z:\tau \mid \Sigma \vdash x : \{y \Rightarrow \sigma\}$, $\texttt{effect}\ g = \{\varepsilon\} \in \sigma$, and $\Gamma, z:\tau \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_2' \cup [x/y]\varepsilon$. By IH, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [l/z][x/y]\varepsilon$. Since y is a free variable, we can select y such that $y \neq x$ and $y \neq z$. We case on if $x = z$:

1. If $z \neq x$, then $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [x/y][l/z]\varepsilon$ By substitution lemma for typing, we have $\Gamma \mid \Sigma \vdash x : \{y \Rightarrow [l/z]\sigma\}$, $\texttt{effect}\ g = [l/z]\varepsilon \in [l/z]\sigma$. Using Subeffect-Def-1, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup \{x.g\}$, which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$
2. If $z = x$ Then we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [l/x, y]\varepsilon$, which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [l/y][l/z]\varepsilon$
   We case on the derivation of $\Gamma, z:\tau \mid \Sigma \vdash z : \{y \Rightarrow \sigma\}$.
   (a) (T-Var)

$$\frac{z:\tau \in \Gamma, z:\tau}{\Gamma, z:\tau \mid \Sigma \vdash z:\tau}$$

   So $\tau = \{y \Rightarrow \sigma\}$. By our assumption, we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$. Since $\texttt{effect}\ g = \{\varepsilon\} \in \sigma$, we have $\texttt{effect}\ g = \{[l/z]\varepsilon\} \in [l/z]\sigma$. Therefore, we can use Subeffect-Def-1 on $\{l.g\}$ to derive $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup \{l.g\}$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$

   (b) (T-Sub)

$$\frac{\Gamma, z:\tau \mid \Sigma \vdash z:\tau_1 \quad \Gamma, z:\tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}}{\Gamma, z:\tau \mid \Sigma \vdash z:\{y \Rightarrow \sigma\}}$$

   Notice that we introduced a new type τ_1 that z can be ascribed to. The judgment $\Gamma, z:\tau \mid \Sigma \vdash z:\tau_1$ can be derived by T-Sub, which introduce a new type τ_2 such that $\Gamma, z:\tau \mid \Sigma \vdash \tau_2 <: \tau_1$, or T-Var, which shows $\tau_1 = \tau$. Therefore if we follow the derivation tree, we get a chain relation $\Gamma, z:\tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}$, $\Gamma, z:\tau \mid \Sigma \vdash \tau_2 <: \tau_1, \ldots, \Gamma, z:\tau \mid \Sigma \vdash \tau <: \tau_n$. We can apply IH on these judgments, so we have a chain $\Gamma \mid \Sigma \vdash [l/z]\tau_1 <: \{y \Rightarrow [l/z]\sigma\}$, $\Gamma \mid \Sigma \vdash$

$[l/z]\tau_2 <: [l/z]\tau_1, \ldots, \Gamma \mid \Sigma \vdash [l/z]\tau <: [l/z]\tau_n$. By transitivity of subtyping, we have $\Gamma \mid \Sigma \vdash [l/z]\tau <: \{y \Rightarrow [l/z]\sigma\}$. So we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$. The rest of the proof is similar to case (a).

Case SUBEFFECT-DEF-2*:* This case is identical to *Case* SUBEFFECT-UPPERBOUND
Case SUBEFFECT-LOWERBOUND*:* This case is identical to *Case* SUBEFFECT-DEF-1
Case WF-EFFECT*:*Let $n_i.g_j \in \varepsilon$ be arbitrary. By inversion, we have $\Gamma, z : \tau \mid \Sigma \vdash n_i : \{\}\{y_i \Rightarrow \overline{\sigma_i}\}$. and the effect declaration of g_j is in $\overline{\sigma_i}$. By IH, we have $\Gamma \mid \Sigma \vdash [l/z]n_i : \{\}\{y_i \Rightarrow [l/z]\overline{\sigma_i}\}$ and the effect declaration of g_j is in $\overline{\sigma_i}$. So we have $[l/z]\varepsilon\ wf$ by WF-Effect.

Thus, substituting terms in a well-typed expression preserves the typing. □

B.2 Proof of Theorem 4 (Preservation)

If $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, $\mu : \Sigma$, and $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$, then $\exists \Sigma' \supseteq \Sigma$, $\mu' : \Sigma'$, $\exists \varepsilon'$, such that $\Gamma \vdash \varepsilon' <: \varepsilon$, and $\Gamma \mid \Sigma' \vdash e' : \{\varepsilon'\}\ \tau$.

Proof. The proof is by induction on a derivation of $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$. At each step of the induction, we assume that the desired property holds for all subderivations and proceed by case analysis on the final rule in the derivation. Since we assumed $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$ and there are no evaluation rules corresponding to variables or locations, the cases when e is a variable (T-VAR) or a location (T-LOC) cannot arise. For the other cases, we argue as follows:

Case T-NEW*:* $e = \texttt{new}(x \Rightarrow \overline{d})$, and by inversion on T-NEW, we get $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \mid \Sigma \vdash d_i : \sigma_i$. The store changes from μ to $\mu' = \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}$, i.e., the new store is the old store augmented with a new mapping for the location l, which was not in the old store ($l \notin dom(\mu)$). From the premise of the theorem, we know that $\mu : \Sigma$, and by the induction hypothesis, all expressions of Γ are properly allocated in Σ. Then, by T-STORE, we have $\mu,\ l \mapsto \{x \Rightarrow \overline{d}\}\ :\ \Sigma,\ l : \{x \Rightarrow \overline{\sigma}\}$, which implies that $\Sigma' = \Sigma,\ l : \{x \Rightarrow \overline{\sigma}\}$. Finally, by T-LOC, $\Gamma \mid \Sigma \vdash l : \{\}\ \{x \Rightarrow \overline{\sigma}\}$, and $\varepsilon' = \varnothing = \varepsilon$. Thus, the right-hand side is well typed.

Case T-METHOD*:* $e = e_1.m(e_2)$, and by the definition of the evaluation relation, there are two subcases:

Subcase E-CONGRUENCE*:* In this case, either $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ or e_1 is a value and $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$. Then, the result follows from the induction hypothesis and T-METHOD.

Subcase E-METHOD*:* In this case, both e_1 and e_2 are values, namely, locations l_1 and l_2 respectively. Then, by inversion on T-METHOD, we get that $\Gamma \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, $\texttt{def}\ \ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$, $\Gamma \mid \Sigma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf$, $\Gamma \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$, and $\varepsilon = \varepsilon_1 \cup \varepsilon_2 \cup [e_1/x][e_2/y]\varepsilon_3$. The store μ does not change, and since T-STORE has been applied throughout, the store is well typed, and thus, $\Gamma \mid \Sigma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2 = e\ :\ \texttt{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2$. Then, by inversion on DT-DEF, we know that $\Gamma,\ x : \tau_1 \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$ and

$\Gamma, x : \tau_1 \mid \Sigma \vdash \varepsilon' <: \varepsilon$. Finally, by the subsumption lemma, substituting locations for variables in e preserves its type, and therefore, the right-hand side is well typed.

Case T-FIELD*:* $e = e_1.f$, and by the definition of the evaluation relation, there are two subcases:

Subcase E-CONGRUENCE*:* In this case, $\langle e_1 \mid \mu \rangle \longrightarrow \langle e_1' \mid \mu' \rangle$, and the result follows from the induction hypothesis and T-FIELD.

Subcase E-FIELD*:* In this case, e_1 is a value, i.e., a location l. Then, by inversion on T-FIELD, we have $\Gamma \mid \Sigma \vdash l : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}$, where $\varepsilon = \varnothing$, and $\texttt{var}\ \ f : \tau \in \overline{\sigma}$. The store μ does not change, and since T-STORE has been applied throughout, the store is well typed, and thus, $\Gamma \mid \Sigma \vdash \texttt{var}\ \ f : \tau = l_1\ :\ \texttt{var}\ \ f : \tau$. Then, by inversion on DT-VARL, we know that $\Gamma \mid \Sigma \vdash l_1 : \{\}\ \tau$ and $\varepsilon' = \varnothing = \varepsilon$, and the right-hand side is well typed.

Case T-ASSIGN*:* $e = (e_1.f = e_2)$, and by the definition of the evaluation relation, there are two subcases:

Subcase E-CONGRUENCE*:* In this case, either $\langle e_1 \mid \mu \rangle \longrightarrow \langle e_1' \mid \mu' \rangle$ or e_1 is a value and $\langle e_2 \mid \mu \rangle \longrightarrow \langle e_2' \mid \mu' \rangle$. Then, the result follows from the induction hypothesis and T-ASSIGN.

Subcase E-ASSIGN*:* In this case, both e_1 and e_2 are values, namely locations l_1 and l_2 respectively. Then, by inversion on T-ASSIGN, we get that $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, $\texttt{var}\ \ f : \tau \in \overline{\sigma}$, $\Gamma \mid \Sigma \vdash l_2 : \{\varepsilon_2\}\ \tau$, and $\varepsilon = \varepsilon_1 = \varepsilon_2 = \varnothing$. The store changes as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}/l_1 \mapsto \{x \Rightarrow \overline{d}\}]\mu$, where $\overline{d}' = [\texttt{var}\ f : \tau = l_2/\texttt{var}\ f : \tau = l]\overline{d}$. However, since T-STORE has been applied throughout and the substituted location has the type expected by T-STORE, the new store is well typed (as well as the old store), and thus, $\Gamma \mid \Sigma \vdash \texttt{var}\ \ f : \tau = l_2\ :\ \texttt{var}\ \ f : \tau$. Then, by inversion on DT-VARL, we know that $\Gamma \mid \Sigma \vdash l_2 : \{\}\ \tau$ and $\varepsilon' = \varnothing$, and the right-hand side is well typed.

Case T-SUB*:* The result follows directly from the induction hypothesis.

Thus, the program written in this language is always well typed. □

B.3 Proof of Theorem 5 (Progress)

If $\varnothing \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ (i.e., e is a closed, well-typed expression), then either

1. e is a value (i.e., a location) or
2. $\forall \mu$ such that $\mu : \Sigma$, $\exists e', \mu'$ such that $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$.

Proof. The proof is by induction on the derivation of $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, with a case analysis on the last typing rule used. The case when e is a variable (T-VAR) cannot occur, and the case when e is a location (T-LOC) is immediate, since in that case e is a value. For the other cases, we argue as follows:

Case T-NEW*:* $e = \texttt{new}(x \Rightarrow \overline{d})$, and by E-NEW, e can make a step of evaluation if the `new` expression is closed and there is a location available that is not in the current store μ. From the premise of the theorem, we know that the expression is closed, and there are infinitely many

available new locations, and therefore, e indeed can take a step and become a value (i.e., a location l). Then, the new store μ' is $\mu, l \mapsto \{x \Rightarrow \overline{d}\}$, and all the declarations in $\overline{d}$ are mapped in the new store.

Case T-METHOD*:* $e = e_1.m(e_2)$, and by the induction hypothesis applied to $\Gamma \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, either e_1 is a value or else it can make a step of evaluation, and, similarly, by the induction hypothesis applied to $\Gamma \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$, either e_2 is a value or else it can make a step of evaluation. Then, there are two subcases:

Subcase $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ *or* e_1 *is a value and* $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$*:* If e_1 can take a step or if e_1 is a value and e_2 can take a step, then rule E-CONGRUENCE applies to e, and e can take a step.

Subcase e_1 *and* e_2 *are values:* If both e_1 and e_2 are values, i.e., they are locations l_1 and l_2 respectively, then by inversion on T-METHOD, we have $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$ and $\texttt{def}\ \ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$. By inversion on T-LOC, we know that the store contains an appropriate mapping for the location l_1, and since T-STORE has been applied throughout, the store is well typed and $l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu$ with $\texttt{def}\ m(y : \tau_1) : \{\varepsilon_3\}\ \tau_2 = e \in \overline{d}$. Therefore, the rule E-METHOD applies to e, e can take a step, and $\mu' = \mu$.

Case T-FIELD*:* $e = e_1.f$, and by the induction hypothesis, either e_1 can make a step of evaluation or it is a value. Then, there are two subcases:

Subcase $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$*:* If e_1 can take a step, then rule E-CONGRUENCE applies to e, and e can take a step.

Subcase e_1 *is a value:* If e_1 is a value, i.e., a location l, then by inversion on T-FIELD, we have $\Gamma \mid \Sigma \vdash l : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}$ and $\texttt{var}\ \ f : \tau \in \overline{\sigma}$. By inversion on T-LOC, we know that the store contains an appropriate mapping for the location l, and since T-STORE has been applied throughout, the store is well typed and $l \mapsto \{x \Rightarrow \overline{d}\} \in \mu$ with $\texttt{var}\ f : \tau = l_1 \in \overline{d}$. Therefore, the rule E-FIELD applies to e, e can take a step, and $\mu' = \mu$.

Case T-ASSIGN*:* $e = (e_1.f = e_2)$, and by the induction hypothesis, either e_1 is a value or else it can make a step of evaluation, and likewise e_2. Then, there are two subcases:

Subcase $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ *or* e_1 *is a value and* $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$*:* If e_1 can take a step or if e_1 is a value and e_2 can take a step, then rule E-CONGRUENCE applies to e, and e can take a step.

Subcase e_1 *and* e_2 *are values:* If both e_1 and e_2 are values, i.e., they are locations l_1 and l_2 respectively, then by inversion on T-ASSIGN, we have $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, $\texttt{var}\ \ f : \tau \in \overline{\sigma}$, and $\Gamma \mid \Sigma \vdash l_2 : \{\varepsilon_2\}\ \tau$. By inversion on T-LOC, we know that the store contains an appropriate mapping for the locations l_1 and l_2, and since T-STORE has been applied throughout, the store is well typed and $l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu$ with $\texttt{var}\ f : \tau = l \in \overline{d}$. A new well-typed store can be created as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}/l_1 \mapsto \{x \Rightarrow \overline{d}\}]\mu$, where $\overline{d}' = [\texttt{var}\ f : \tau = l_2/\texttt{var}\ f : \tau = l]\overline{d}$. Then, the rule E-ASSIGN applies to e, and e can take a step.

Case T-SUB*:* The result follows directly from the induction hypothesis.

Thus, the program written in this language never gets stuck. □

Appendix C

Type Safety Theorems for Algebraic Effects and Handlers

C.1 Lemmas

Lemma 21. (Substitution)
If $\Gamma, x_j : \tau' \vdash c_i : \sigma$ and $\Gamma \vdash e_j : \tau'$, then $\Gamma \vdash \{e_j/x_j\}c_i : \sigma$, and
If $\Gamma, x_j : \tau' \vdash e_i : \tau$ and $\Gamma \vdash e_j : \tau'$, then $\Gamma \vdash \{e_j/x_j\}e_i : \tau$

Proof. By rule induction on $\Gamma \vdash e : \tau$ and $\Gamma \vdash c : \sigma$

(T-Unit) Trivial
(T-Var) Trivial
(T-Lam)

$$\frac{\Gamma, x_j : \tau', x_i : \tau \vdash c_i : \sigma}{\Gamma, x_j : \tau' \vdash (\lambda x_i : \tau.\ c_i) : \tau \to \sigma}\ \text{(T-LAM)}$$

By IH, we have $\Gamma, x_i : \tau \vdash \{e_j/x_j\}c_i : \sigma$
Then by (T-Lam) we have $\Gamma \vdash (\lambda x_i : \tau.\ \{e_j/x_j\}c_i) : \sigma$.
Which is equivalent to $\Gamma \vdash \{e_j/x_j\}(\lambda x_i : \tau.\ c_i) : \sigma$.
(T-EmbedExp) By inversion and IH
(T-Ret) Follows by induction hypothesis
(T-Op)

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau \quad op \in \Delta_i(\varepsilon)}{\Gamma \vdash op(e_i; y_i.c_i) : \{\varepsilon\}\tau}\ \text{(T-OP)}$$

By inversion we have $\Gamma, x_j : \tau' \vdash e_i : \tau_A$ and $\Gamma, x_j : \tau', y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau$.
Since we can make y_i a fresh variable, we have $\Gamma, y_i : \tau_B, x_j : \tau' \vdash c_i : \{\varepsilon\}\tau$.
Then by IH we have $\Gamma \vdash \{e_j/x_j\}e_i : \tau_A$ and $\Gamma, y_i : \tau_B \vdash \{e_j/x_j\}c_i : \{\varepsilon\}\tau$.
By (T-Op) we have $\Gamma \vdash op(\{e_j/x_j\}e_i; y_i.\{e_j/x_j\}c_i) : \{\varepsilon\}\tau$
Therefore we have $\Gamma \vdash \{e_j/x_j\}(op(e_i; y_i.c_i)) : \{\varepsilon\}\tau$

(T-Seq)

$$\frac{\Gamma, x_j : \tau'' \vdash c_i : \{\varepsilon\}\tau \quad \Gamma, x_j : \tau'', x_i : \tau \vdash c_i' : \{\varepsilon\}\tau'}{\Gamma, x_j : \tau'' \vdash \texttt{ do } x_i \leftarrow c_i \texttt{ in } c_i' : \{\varepsilon\}\tau'} \text{ (T-SEQ)}$$

By IH, we have $\Gamma \vdash \{e_j/x_j\}c_i : \{\varepsilon\}\tau$
Since we can choose x_i as a fresh variable, we have $\Gamma, x_i : \tau, x_j : \tau'' \vdash c_i' : \{\varepsilon\}\tau'$
Then by IH we have $\Gamma, x_i : \tau \vdash \{e_j/x_j\}c_i' : \{\varepsilon\}\tau'$
Then the result follows by (T-Seq)

(T-App) Follows directly by applying IH.

(T-Handle)

$$\frac{\begin{array}{c} h_i = \{\ \texttt{return}\ x \mapsto c^r, op^1(x;k) \mapsto c^1, \ldots, op^n(x;k) \mapsto c^n\} \\ \Gamma, x_j : \tau', x : \tau_A \vdash c^r : \{\varepsilon'\}\tau_B \\ \left\{\Sigma(op^i) = \tau_i \to \tau_i' \quad \Gamma, x_j : \tau', x : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c^i : \{\varepsilon'\}\tau_B\right\}_{1 \le i \le n} \\ \Gamma, x_j : \tau' \vdash c_i : \{\varepsilon\}\tau_A \quad \varepsilon \setminus \{op^i\}_{1 \le i \le n} \subseteq \varepsilon' \end{array}}{\Gamma, x_j : \tau' \vdash \texttt{ with } h_i \texttt{ handle } c_i : \{\varepsilon'\}\tau_B} \text{ (T-HANDLE)}$$

Then handling clauses bind variables x and k in the handling computation c^i, so we can make them fresh variables that do not appear in context Γ. Then we can apply IH to typing judgements in the premise.

(T-Embed) Follows by applying IH

(T-EmbedOp) The proof is similar to the case for (T-Op)

□

Lemma 22. If $\Gamma \vdash c_i : \{\varepsilon\}\tau$ then $\overline{\Delta_i}(\varepsilon) = \varepsilon$

Proof. By induction on derivation of $\Gamma \vdash c : \sigma$. (T-Ret) has a premise the ensures the lemma is correct. For other rules, the result is immediate by applying IH.

□

Lemma 23. If $\varepsilon \le_l \varepsilon'$, then $\varepsilon \setminus op \le_l \varepsilon' \setminus op$

Proof. By induction on $\varepsilon \le_l \varepsilon$. The proof is straightforward. □

Lemma 24. If $op \le_l \varepsilon$, then $op \le_l \varepsilon \setminus op'$

Proof. By induction on the derivation of $\varepsilon \le_l \varepsilon$. If (R-Eff1) is used, then the proof is straightforward because the subset relation on the premise still holds. If (R-Eff2) is used, by inversion on (R-Eff2), we have $op \le_l \varepsilon'$ and $\varepsilon' \le_{l'} \varepsilon$. By IH we have $op \le_l \varepsilon' \setminus op'$. By lemma 23 we have $\varepsilon' \setminus op' \le_{l'} \varepsilon \setminus op'$. Then the result follows by (R-Eff2) □

Lemma 25. If $\tau \le_l \tau'$ then $\tau' \le_{rev(l)} \tau$

Proof. By induction on the type relation rules. The proof consists of simple arguments that follow directly from IH. □

C.2 Preservation

Proof of lemma 6 (Preservation for expressions)

For all agent i, If $\Gamma \vdash e_i : \tau$ and $e_i \mapsto e_i'$, then $\Gamma \vdash e_i' : \tau$.

Proof. By induction on derivation of $e_i \mapsto e_i'$

(E-Congruence) By inversion on the typing rule for embedded expressions, we have $\Gamma \vdash e_j : \tau'$. By IH, we have $\Gamma \vdash e_j' : \tau'$. Then we use (E-Contruence) to derive $\Gamma \vdash [e_j']_j^\tau : \tau$

(E-Unit) Follows immediately from (T-Unit)

(E-Lambda) By inversion on (T-Embed), we have $\Gamma \vdash \lambda x_j : \tau'.\ c_j : \tau' \to \sigma'$, where $\tau' \to \sigma' \leq_{ji} \tau \to \sigma$.
By inversion on (R-Arrow), we have $\tau' \leq_{ji} \tau$ and $\sigma' \leq_{ji} \sigma$
By inversion on (T-Lambda), we have $\Gamma, x_j : \tau' \vdash c_j : \sigma'$. And since x_i is a fresh variable in c_j, we have $\Gamma, x_i : \tau, x_j : \tau' \vdash c_j : \sigma'$
By lemma 25, we have $\tau \leq_{ij} \tau'$, and therefore $\Gamma, x_i : \tau \vdash [x_i]_i^{\tau'} : \tau'$
Then we can use the substitution lemma to derive $\Gamma, x_i : \tau \vdash \{[x_i]_i^{\tau'}/x_j\}c_j : \sigma'$.
Then by (T-Embed), we have $\Gamma, x_i : \tau \vdash [\{[x_i]_i^{\tau'}/x_j\}c_j]_j^\sigma : \sigma$
Then the result follows by (T-Lambda).

□

Proof of lemma 7 (Preservation for computations)

If $\Gamma \vdash c_i : \{\varepsilon\}\tau$ and $c_i \longrightarrow c_i'$, then $\Gamma \vdash c_i' : \{\varepsilon\}\tau$

Proof. (Sketch) By induction on the derivation that $c_i \longrightarrow c_i'$. We proceed by the cases on the last step of the derivation.

1. E-Ret: By inversion, $\Gamma \vdash e_i : \tau$. By preservation of expressions and IH, we have $\Gamma \vdash e_i' : \tau$. Then we can use E-Ret to derive $\Gamma \vdash c_i' : \{\varepsilon\}\tau$
2. E-Op: Follow immediately from inversion and IH
3. E-EmbedOp1: Follow immediately from inversion and IH
4. E-EmbedOp2:

$$\frac{\overline{\Delta_i}(\varepsilon) = \varepsilon''}{[op_j]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow [op_j]_l^{\varepsilon''}(v_i; y_i.c_i)}\ \text{(E-EMBEDOP2)}$$

We have the typing rule as follows:

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]_l^\varepsilon(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}$$

Since $\overline{\Delta_i}(\varepsilon'') = \varepsilon''$ and $\varepsilon'' = \overline{\Delta_i}(\varepsilon)$, we have $\overline{\Delta_i}(\varepsilon'') \subseteq \overline{\Delta_i}(\varepsilon')$. Then we can use T-EmbedOp to derive $\Gamma \vdash [op]_l^{\varepsilon''}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau$

5. E-EmbedOp3: We have the typing rule as follows:

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]_l^{\varepsilon}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}$$

   By E-EmbedOp3, $op \in \overline{\Delta_i}(\varepsilon)$. So $op \in \overline{\Delta_i}(\varepsilon')$. By inversion on the typing rule, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau$. By lemma 22, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\overline{\Delta_i}(\varepsilon')\}\tau$. Then we can use T-Op to derive the designed result $\Gamma \vdash op(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau$

6. E-EmbedOp4: We have the typing rule as follows:

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]_l^{\varepsilon}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}$$

   By lemma 24, we have $op \leq_{li} \varepsilon \setminus op'$. By inversion on the typing rule, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau$ and $\varepsilon \subseteq \overline{\Delta_i}(\varepsilon')$. So $\varepsilon \setminus op' \subseteq \overline{\Delta_i}(\varepsilon')$. By lemma 22, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\overline{\Delta_i}(\varepsilon')\}\tau$. Then we can apply T-EmbedOp again to derive $\Gamma \vdash [op]_l^{\varepsilon \setminus op'}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau$

7. E-App1: Follows immediately by T-App
8. E-App2: Follows immediately by T-App
9. E-App3: By inversion of T-App, we $\Gamma \vdash (\lambda x_i : \tau.\ c_i) : \tau \to \sigma, \Gamma \vdash v_i : \tau$. By inversion of T-Lam, $\Gamma, x_i : \tau \vdash c_i : \sigma$. By substitution lemma, we have $\Gamma \vdash \{v_i/x_i\}c_i : \sigma$.
10. E-Seq1: Follows immediately by T-Seq and IH.
11. E-Seq2: By inversion on T-Seq, we have $\Gamma \vdash \ \texttt{return}\ v_i : \{\varepsilon\}\tau$ and $\Gamma, x_i : \tau \vdash c_i' : \{\varepsilon\}\tau'$. By inversion on T-Ret, we have $\Gamma \vdash v_i : \tau$. Then by substitution lemma we have $\Gamma \vdash \{v_i/x\}c_i' : \{\varepsilon\}\tau'$.
12. E-Seq3:

$$\frac{}{\texttt{do}\ x \leftarrow op_i(v_i; y_i.c_i)\texttt{in}\ c_i' \longrightarrow op_i(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')}\ \text{(E-SEQ3)}$$

    By inversion of T-Seq, we have $\Gamma \vdash op_i(v_i; y_i.c_i) : \{\varepsilon\}\tau$ and $\Gamma, x : \tau \vdash c_i' : \{\varepsilon\}\tau'$. By inversion on T-OP, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau$ and $op \in \varepsilon$ and $\Gamma \vdash v_i : \tau_A$. Then by T-Seq, we have $\Gamma, y_i : \tau_B \vdash \ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon\}\tau'$. Then we can use T-Op to derive $\Gamma \vdash op_i(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i') : \{\varepsilon\}\tau'$.

13. E-Seq4

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{do}\ x \leftarrow [op_j]_l^{\varepsilon}(v_i; y_i.c_i)\texttt{in}\ c_i' \longrightarrow [op_j]_l^{\varepsilon}(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')}\ \text{(E-SEQ4)}$$

$$\frac{\Gamma \vdash c_i : \{\varepsilon'\}\tau \quad \Gamma, x_i : \tau \vdash c_i' : \{\varepsilon'\}\tau'}{\Gamma \vdash \ \texttt{do}\ x_i \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon'\}\tau'}\ \text{(T-SEQ)}$$

    By inversion on T-Seq, we have $\Gamma \vdash [op_j]_l^{\varepsilon}(v_i; y_i.c_i) : \{\varepsilon'\}\tau$ and $\Gamma, x : \tau \vdash c_i' : \{\varepsilon'\}\tau'$. Then by inversion on T-EmbedOp, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau$, $\overline{\Delta_i}(\varepsilon) \subseteq \varepsilon'$. Then by T-Seq, we have $\Gamma, y_i : \tau_B \vdash \ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon'\}\tau'$. Then by T-EmbedOp, we have $\Gamma \vdash [op]_l^{\varepsilon}(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i') : \{\varepsilon'\}\tau'$

14. E-Handle1: Follows immediately by inversion on T-Handle and IH
15. E-Handle2: By T-Handle, we have $\Gamma \vdash \texttt{with}\ h_i\ \texttt{handle}\ \texttt{return}\ v_i : \{\varepsilon'\}\tau_B$. By inversion on T-Handle, we have $\Gamma, x_i : \tau_A \vdash c_i' : \{\varepsilon\}\tau_B$, and $\Gamma \vdash \texttt{return}\ v_i : \{\varepsilon\}\tau_A$. By inversion on T-Ret, we have $\Gamma \vdash v_i : \tau_A$. Then by substitution lemma, we have $\Gamma \vdash \{v_i/x_i\}c_i' : \{\varepsilon'\}\tau_B$.
16. E-Handle3

$$\frac{op(x_i; k) \mapsto c_i' \in h_i \quad \Sigma(op) = \tau_i \to \tau_i'}{\texttt{with}\ h_i\ \texttt{handle}\ op(v; y_i.c_i) \longrightarrow \{v_i/x_i\}\{(\lambda y_i : \tau_i'.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i)/k\}c_i'}$$

By T-Handle, we have $\Gamma \vdash \texttt{with}\ h_i\ \texttt{handle}\ op(v; y_i.c_i) : \{\varepsilon'\}\tau_B$. By inversion on T-Handle, we have $\Gamma, x_i : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c_i' : \{\varepsilon'\}\tau_B$, and $\Gamma \vdash op(v; y_i.c_i) : \{\varepsilon\}\tau_A$. By inversion on T-Op, we have $\Gamma \vdash v_i : \tau_i$ and $\Gamma, y_i : \tau_i' \vdash c_i : \{\varepsilon\}\tau_A$. By T-Handle, we have $\Gamma, y_i : \tau_i' \vdash \texttt{with}\ h_i\ \texttt{handle}\ c_i : \{\varepsilon'\}\tau_B$. Then by T-Lam, we have $\Gamma \vdash \lambda y_i : \tau_i'.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i : \tau_i' \to \{\varepsilon'\}\tau_B$. Then, by substitution lemma, we have $\Gamma \vdash \{v_i/x_i\}\{(\lambda y_i : \tau_i'.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i)/k\}c_i' : \{\varepsilon'\}\tau_B$.
17. E-Handle4:

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{with}\ h_i\ \texttt{handle}\ [op]_l^{\varepsilon}(v_i, y_i.c_i) \longrightarrow [op]_l^{\varepsilon}(v_i; y_i.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i))}\ \text{(E-HANDLE4)}$$

$$\frac{\begin{array}{c} h_i = \{\ \texttt{return}\ x \mapsto c^r, op^1(x; k) \mapsto c^1, \ldots, op^n(x; k) \mapsto c^n\} \\ \Gamma, x : \tau_A \vdash c^r : \{\varepsilon'\}\tau_B \\ \left\{\Sigma(op^i) = \tau_i \to \tau_i' \quad \Gamma, x : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c^i : \{\varepsilon'\}\tau_B\right\}_{1 \le i \le n} \\ \Gamma \vdash c_i : \{\varepsilon''\}\tau_A \quad \varepsilon'' \setminus \{op^i\}_{1 \le i \le n} \subseteq \varepsilon' \end{array}}{\Gamma \vdash \texttt{with}\ h_i\ \texttt{handle}\ c_i : \{\varepsilon'\}\tau_B}\ \text{(T-HANDLE)}$$

By T-Handle, we have $\Gamma \vdash \texttt{with}\ h_i\ \texttt{handle}\ [op]_l^{\varepsilon}(v; y_i.c_i) : \{\varepsilon'\}\tau_B$. By inversion on T-Handle, we have $\Gamma \vdash [op]_l^{\varepsilon}(v; y_i.c_i) : \{\varepsilon''\}\tau_A$ and $\varepsilon'' \setminus \{op^i\} \subseteq \varepsilon'$. By inversion on T-EmbedOp, we have $\Gamma \vdash v_i : \tau_i$, $\Gamma, y_i : \tau_i' \vdash c_i : \{\varepsilon''\}\tau_A$ and $\varepsilon \subseteq \varepsilon''$. Since ε doesn't contain any concrete operation, we have $\varepsilon \subseteq \varepsilon'' \setminus \{op^i\} \subseteq \varepsilon'$. Then by T-Handle, we have $\Gamma, y_i : \tau_i' \vdash \texttt{with}\ h_i\ \texttt{handle}\ c_i : \{\varepsilon'\}\tau_B$. Then, we use T-EmbedOp to derive $\Gamma \vdash [op]_l^{\varepsilon}(v_i; y_i.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i) : \{\varepsilon'\}\tau_B$
18. E-Embed1: Follows immediately from Inversion and IH
19. E-Embed2: By typing rule, we have $\Gamma \vdash [\ \texttt{return}\ v_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$. By inversion on the typing rule, we have $\Gamma \vdash \texttt{return}\ v_j : \{\varepsilon'\}\tau'$ such that $\{\varepsilon\}\tau' \le_{li} \{\varepsilon\}\tau$. By inversion on R-Sigma, we have $\tau' \le_{li} \tau$. Then by T-EmbedExp, we have $\Gamma \vdash [v_j]_l^{\tau} : \tau$. Then by T-Ret, we have $\Gamma \vdash \texttt{return}\ [v_j]_l^{\tau} : \{\varepsilon\}\tau$. $\Gamma \vdash [\ \texttt{return}\ v_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$
20. E-Embed3:

$$\frac{\Sigma(op) = \tau_A \to \tau_B}{[op(v_j; y_j.c_j)]_l^{\{\varepsilon\}\tau} \longrightarrow [op]_l^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau})}\ \text{(E-EMBED3)}$$

By typing rule, we have $\Gamma \vdash op(v_j; y_j.c_j) : \{\varepsilon'\}\tau'$, where $\{\varepsilon'\}\tau' \leq_{li} \{\varepsilon\}\tau$. By inversion on T-Op, we have $\Gamma \vdash v_j : \tau_A$, and $\Gamma, y_j : \tau_B \vdash c_j : \{\varepsilon'\}\tau'$. Then, by T-EmbedExp, we have $\Gamma \vdash [v_j]_j^{\tau_A} : \tau_A$. By substitution lemma, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}c_j : \{\varepsilon'\}\tau'$. By T-Embed, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$. Then we can use T-EmbedOp to derive $\Gamma \vdash [op]_l^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau}) : \{\varepsilon\}\tau$.

21. E-Embed4:

$$\frac{\Sigma(op_k) = \tau_A \to \tau_B \quad \Delta_j(\varepsilon') = \varepsilon' \quad op \notin \varepsilon'}{[[op_k]_{l'}^{\varepsilon'}(v_j; y_j.c_j)]_l^{\{\varepsilon\}\tau} \longrightarrow [op_k]_{l'jl}^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau})} \text{ (E-EMBED4)}$$

By typing rule, we have $\Gamma \vdash [op_K]_{l'}^{\varepsilon'}(v_j; y_j.c_j) : \{\varepsilon''\}\tau''$, where $\{\varepsilon''\}\tau'' \leq_{li} \{\varepsilon\}\tau$. By inversion on T-EmbedOp, we have $\Gamma \vdash v_j : \tau_A$ and $\Gamma, y_j : \tau_B \vdash c_j : \{\varepsilon''\}\tau''$. Then, by T-EmbedExp, we have $\Gamma \vdash [v_j]_j^{\tau_A} : \tau_A$. By substitution lemma, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}c_j : \{\varepsilon''\}\tau''$. By T-Embed, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$. Then we use T-EmbedOp to derive $\Gamma \vdash [op_k]_{l'jl}^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau}) : \{\varepsilon\}\tau$.

□

C.3 Progress

Proof of lemma 8 (Progress for expressions)

For agent i, if $\varnothing \vdash e_i : \tau$ then either $e_i = v_i$ or $e_i \longrightarrow e_i'$.

Proof. By induction on structure of e_i.

Case $e_i = ()$: e_i is already a value.
Case $e_i = \lambda x : \tau.c$: e_i is already a value.
Case $e_i = [e_j]_j^{\tau}$: By IH, either e_j is a j-value, or $e_j \longrightarrow e_j'$. If $e_j \longrightarrow e_j'$, then by (E-Congruence), $[e_j]_j^{\tau} \longrightarrow [e_j']_j^{\tau}$. If e_j is a value, then it is either $()$ or $\lambda x_j : \tau'.c_j$. So e_i can be evaluated by (E-Unit) and (E-Lambda) correspondingly.

□

Proof of lemma 9 (Progress for computation)

If $\varnothing \vdash c_i : \{\varepsilon\}\tau$ then either

1. $c_i \longrightarrow c_i'$
2. $c_i =$ `return` v_i
3. $c_i = op(v_i; y_i.c_i')$
4. $c_i = [op]_l^{\varepsilon}(v_i; y_i.c_i')$ and $op \notin \varepsilon$

Proof. By induction on structure of c_i.

Case $c_i =$ `return` e_i : Immediate by applying IH on e_i.

Case $c_i = op(e_i, y_i.c_i')$: Immediate by applying IH on e_i.
Case $c_i = [c_j]_j^\sigma$: By IH on c_j, c_j can either evaluates to another computation, or be a return statement, an operation call, or an embedded operation call. Then we can apply (E-Embed) rules to evaluate c_i accordingly.
Case $c_i = [op]_l^\varepsilon(e_i, y_i.c_i')$: Follows directly by applying IH on e_i.
Case $c_i = e_i\ e_i'$: If e_i or e_i' are not values, then (E-App1) or (E-App2) can be applied to c_i. Otherwise, (E-App3) could be applied.
Case $c_i = \texttt{do}\ x \leftarrow c_i'\ \texttt{in}\ c_i''$: Follows directly from applying IH on c_i'.
Case $c_i = \texttt{with}\ h_1\ \texttt{handle}\ c_i'$: Follows directly from applying IH on c_i'.

□