Extending Abstract Effects with Bounds and Algebraic Handlers

Anlun Xu

November 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

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

Keywords: Effect Systems

For my dog

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), which takes effects that are already built into the language–such as reading and writing state or exceptions–and provides a way to restrict them. And 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 system, 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 softwares. The first part of our work is a restrictive path-dependent effect system that provide a granular effect hierarchy by allowing abstract effect members to be bounded. We present a full formalization of the effect-system, and provide 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. We give a formalization of the system and provide proofs for type soundness and dynamic semantic correctness of abstract algebraic effects.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An effect system can be used to reason about the side effects of code, such as reads and writes to memory, exceptions, and I/O operations. Java's checked exceptions is a simple effect system that has found widespread use, and interest is growing in effect systems for reasoning about security [22], memory effects [14], and concurrency [3, 4, 6].

Requirements for a scalable effect system Unfortunately, effect systems have not been widely adopted, other than checked exceptions in Java, a feature that is widely viewed as problematic [23]. The root of the problem is that existing effect systems do not provide adequate support for scaling to programs that are larger and have complex structure. Any adequate solution must support *effect abstraction*, *effect composition*, and *path-dependent effects*. Furthermore, effects should be an inherent part of the type system, instead of being encoding of other type abstractions such as monad.

Abstraction is key to achieving scale in general, and a principal form of abstraction is abstract types [16], a modern form of which appears as abstract type members in Scala [17]. 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 an abstraction. In order to integrate with modularity mechanisms, and by analogy to type members, we define effects using *effect members* of modules or objects. For example, a `file.Read` effect could

abstract a lower-level `system.FFI` effect. Then clients of a file should be able to reason about side effects in terms of file reads and writes, not in terms of the low-level calls that are made to the foreign function interface (FFI). 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).

Effect polymorphism is a form of parametric polymorphism that allows functions or types to be implemented generically for handling computations with different effects [14]. 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 general 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.

We also leverage *path-dependent effects*, i.e., effects whose definitions depend on an object. This adds expressiveness; for example, if we have two `File` objects, `x` and `y`, we can distinguish effects on one file from effects on the other: the effects `x.Read` and `y.Read` are distinct. Path-dependent effects are particularly important in the context of modules, where two different modules may implement the same abstract effect in different ways. For example, it may be important to distinguish `db1.Query` from `db2.Query` if `db1` is an interface to a database stored in the local file system whereas `db2` is a database accessed over the network.

Effects should be an inherent part of the type system, instead of being encoded as a type abstraction. The reason is twofold: although effect checking may be implemented in terms of monad, the safety of the effect system is compromised. The effectful code can bypass the effect-checking process if the programmer use the effectful code in a context which is not encapsulated

by a monad. The other reason is that monad could be unintuitive to use for programmers outside of the functional programming community, while a type system enhanced by effect types doesn't require prior knowledge of monad and is straightforward to use.

Design of the effect system in Wyvern This paper presents a novel and scalable effect-system design that supports effect abstraction and composition. 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.

Just as Scala's type members can be used to encode parametric polymorphism over types, our effect members double as a way to provide effect polymorphism. Bounded effect polymorphism is also provided in our system, because abstract effect members can be bounded by upper or lower bounds. We follow numerous prior Scala formalisms in including polymorphism via this encoding rather than explicitly; this keeps the formal system simpler without giving up expressive power.

Finally, because effect members are defined on objects, our effects are *generative*, even dynamically. This yields great expressivity: each object created at runtime defines a new effect for each effect member in that object so that, for example, we can separately track effects on different `File` objects, statically distinguishing the effects on one object from the effects on another.

Evaluation and Security Applications. A promising area of application for effects is software security. For example, in the setting of mobile code, [22] proposed that effects could be used to ensure that any untrusted code we download can only access the system resources it needs to do its tasks, thus following the principle of least privilege [5]. We are not aware of prior work that

explores this idea in depth.

In order to evaluate our design for effect abstraction, we have incorporated it into an effect system that tracks the use of system resources such as the file system, network, and keyboard. Our effect system is intended to help developers reason about which source code modules use these resources. Through the use of abstraction, we can "lift" low-level resources such as the file system into higher-level resources such as a logging facility or a database and enable application code to reason in terms of effects on those higher-level resources when appropriate. In fact, even the use of resources such as the file system is scaffolded as an abstraction on top of a primitive `system.FFI` effect that our system attaches to uses of the language's foreign function interface. A set of illustrative examples demonstrates the benefits of abstraction for effect aggregation, as well as for information hiding and software evolution. Finally, we show how our effect system allows us to reason about the *authority* [15] of code, i.e., what effects a component can have, as well as the *attenuation* of that authority.

Our effect system is implemented in the context of Wyvern, a programming language designed for highly productive development of secure software systems. In this paper, we give several concrete examples of how our effect-system design can be used in software production, all of which are functional Wyvern code that runs in the Wyvern regression test suite.

Outline and Contributions. The next section introduces a running example, after which we describe the main contributions of our paper:

- The design of a novel effect system fulfilling the requirements above. Our system is the first to bring together effect abstraction and composition with the effect member construct. Ours is also 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 3.1);
- The application of our effect system to a number of forms of security reasoning, illustrating its expressiveness and making the benefits described above concrete (Section **??**);

- 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 (Section 3.3);
- A formalization of authority using effects, and of authority attenuation (Section **??**);
- A feasibility demonstration, via the implementation of our approach in the Wyvern programming language (Section **??**).

The last sections in the paper discuss related work and conclude.

Chapter 2

Background and Motivation

2.1 Descriptive Effect System

Denotational vs. Descriptive Effects. Filinski [7] makes a distinction between two strands of work on effects. A *denotational* approach, which includes algebraic effects, defines the semantics of computational effects based on primitives. A *restrictive* approach (e.g., Java's checked exceptions) takes effects that are already built into the language–such as reading and writing state or exceptions–and provides a way to restrict them.

Origins of Effect Systems. Effect Systems were originally proposed by Lucassen [13] to track reads and writes to memory, and then Lucassen and Gifford [14] extended this effect system to support polymorphism. Effects have since been used for a wide variety of purposes, including exceptions in Java [9] and asynchronous event handling [4]. Turbak and Gifford [22] previously proposed effects as a mechanism for reasoning about security, which is the main application that we discuss.

Prior Work on Bounded Effect Polymorphism. A limited form of bounded effect polymorphism were explored by Trifonov and Shao [21], who bound effect parameters by the resources they may act on; however, the bound cannot be another aribrary effect, as in our system. Long et al. [12] use a form of bounded effect polymorphism internally but do not expose it to users of

their system.

Path-Dependent Effects JML's data groups [11] have some superficial similarities to Wyvern's effect members. Data groups are identifiers bound in a type that refer to a collection of fields and other data groups. They allow a form of abstract reasoning, in that clients can reason about reads and writes to the relevant state without knowing the underlying definitions. Data groups are designed specifically to capture the modification of state, and it is not obvious how to generalize them to other forms of effects.

The closest prior work on path-dependent effects, by Greenhouse and Boyland [8], allows programmers to declare regions as members of types; this supports a form of path-dependency in read and write effects on regions. Our formalism expresses path-dependent effects based on the type theory of DOT [1], which we find to be cleaner and easier to extend with the unique bounded abstraction features of our system. Amin et al.'s type members can be left abstract or refined by upper or lower bounds, and were a direct inspiration for our work on bounded abstract effects.

Subeffecting. Some effect systems, such as Koka [10], provide a built-in set of effects with fixed sub-effecting relationships between them. Rytz et al. [20] supports 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.

2.2 Algebraic Effects

Algebraic Effects, Generativity, and Abstraction. Algebraic effects and handlers [18, 19] are a way of implementing certain kinds of side effects such as exceptions and mutable state in an otherwise purely functional setting. As described above, algebraic effects fall into the "denotational" rather than "descriptive" family of effects work; these lines of work are quite

divergent, and it is often unclear how to translate technical ideas from one setting to the other. However, certain papers explore parallels to our work, despite the major contextual differences.

Bračevac et al. [4] use algebraic effects to support asynchronous, event-based reactive programs. They need to use a different algebraic effect for each join operation that correlates events; thus, they want effects to be generative. This generativity is at a per-module level, however, whereas our work supports per-object generativity.

Zhang and Myers [24] describe a design for algebraic effects that preserves abstraction in the sense 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.

Biernacki et al. [2] discuss how to abstract algebraic effects using existentials. The setting of algebraic effects makes their work quite different from ours: their abstraction hides the "handler" of an effect, which is a dynamic mechanism that actually implements effects such as exceptions or mutable state. In contrast, our work allows a high-level effect to be defined in terms of zero or more lower-level effects, and our abstraction mechanism allows the programmer to hide the lower-level effects that constitute the higher-level effect. Our system, unlike algebraic effect systems, is purely static. We do not attempt to implement effects, but rather give the programmer a system for reasoning about side effects on system resources and program objects. It is not clear that defining a high-level effect that encapsulates multiple low-level events is sensible in the setting of algebraic effects, since this would require merging effect implementations that could be as diverse as mutable state and exception handling. It is also not clear how Biernacki et al.'s abstraction of algebraic effects could apply to the security scenarios we examine in Section **??**, since some of our scenarios rely critically on abstracting lower-level events as higher-level ones.

Chapter 3

Bounded Abstract Effects

3.1 Wyvern Effect Basics

Consider the code in Fig. 3.1 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.[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. 3.2) 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

[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(): {ReadLog} String = f.read()
def updateLog(newEntry: String): {UpdateLog} Unit = f.append(newEntry)
```

Figure 3.1: 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 3.2: The type of the file resource.

`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.

3.1.1 Path-dependent Effects

Effects are members of objects[2], so we refer to them with the form `variable.EffectName`, where `variable` is a 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.

[2]Modules are an important special case of objects

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 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 specify 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.

3.1.2 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 (more on this in Section **??**).

3.1.3 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 annota-

tions, 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.

3.1.4 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 3.1.2, the `system.FFI` effect is an effect that describes function calls though 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.

3.2 Effect Bounds

Our effect system also 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. 3.1, except the `ReadLog` and `UpdateLog` effects are upper-bounded by the corresponding effects in the `fileSystem` module:

```
resource type BoundedLogger
   effect ReadLog <= {fileSystem.Read}
   effect UpdateLog <= {fileSystem.Append}
   ... // same as in the type Logger in Fig. 2
```

Any object implementing 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 and 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 [**?**] 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

[8] 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 [**?**], 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 3.3: Wyvern's object-oriented core syntax.

Fig. 3.3 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

```
$\keyw{let} \mathit{logger} = \keywadj{new}(x \Rightarrow$
$\quad\keyw{def} \mathit{apply}(\mathit{f} : \mathit{File}) : \{ \}~\mathit{
    Logger}$
$\quad\qquad\keywadj{new} (\_ \Rightarrow$
$\qquad\qquad\keyw{effect} \mathit{ReadLog} = \{ \mathit{f}.\mathit{Read} \}
    $
$\qquad\qquad\keyw{effect} \mathit{UpdateLog} = \{ \mathit{f}.\mathit{Append
    } \}$
$\qquad\qquad\keyw{def} \mathit{readLog}() : \{ \mathit{ReadLog} \}~\mathit{
    String} = \mathit{f}.\mathit{read}()$
$\qquad\qquad\keyw{def} \mathit{updateLog}(\mathit{newEntry} : \mathit{
    String}) : \{ \mathit{UpdateLog} \}~\mathit{Unit} = \mathit{f}.\mathit{
    append}(\mathit{newEntry})))$
$\keyw{in} ... \mathit{//}~\mathit{calls}~\mathit{logger.apply(...)}$
```

Figure 3.4: A simplified translation of the `logger` module from Fig. 3.1 into Wyvern's object-oriented core.

Fig. 3.4 presents a simplified translation of the `logger` module from Fig. 3.1 into Wyvern's object-oriented core (for a full description of the translation mechanism, refer to [**?**]). For our 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 type check expressions, we must make sure that effects and types are well formed. Wyvern well-formedness rules are mostly straightforward and are shown in Fig. 3.5. 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. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.5: Wyvern well-formedness rules.

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

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

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

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

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

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

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

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

Figure 3.6: Wyvern static semantics.

3.3.4 Static Semantics

Wyvern's static semantics is presented in Fig. 3.6. 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 3.7: Wyvern subeffecting rules.

As we already saw in the T-SUB, and DT-DEF rules above and as we will see more in the upcoming Section 3.3.5, to compare two sets of effects, we use subeffecting rules, which are presented in Fig. 3.7. If an effect is a subset of another effect, 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

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

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

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

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

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

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

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

Figure 3.8: Rules for determining the size of effect definitions.

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. 3.8) is finite.*

Proof. By rules Safe-1, Safe-2, Safe-3, and Safe-4 in Fig. 3.5, 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, 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.

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

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

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

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

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

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

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

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

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

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

Figure 3.9: Wyvern subtyping rules.

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. 3.9. 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 3.10: 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 emissible 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

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

Figure 3.11: Wyvern's object-oriented core syntax with dynamic forms.

Fig. 3.11 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.

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

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

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

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

$\boxed{\mu : \Sigma}$

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

Figure 3.12: Wyvern static semantics affected by dynamic semantics.

Dynamic Semantics

The dynamic semantics that we use for Wyvern's effect system is shown in Fig. 3.13 and is similar to the one described in prior work [**?**]. 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 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 **??**.

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

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

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

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

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

Figure 3.13: Wyvern dynamic semantics.

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* $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$*.*

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. 2.1, 3.1.1

[2] Dariusz Biernacki, Maciej Piròg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting Algebraic Effects. In *Symposium on Principles of Programming Languages*, 2019. 2.2

[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] 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, 2.2

[5] Peter J. Denning. Fault Tolerant Operating Systems. *ACM Comput. Surv.*, 8(4):359–389, December 1976. ISSN 0360-0300. doi: 10.1145/356678.356680. URL http://doi.

acm.org/10.1145/356678.356680. 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. 2.1

[8] Aaron Greenhouse and John Boyland. An object-oriented effects system. pages 205–229, 06 1999. doi: 10.1007/3-540-48743-3_10. 2.1, 3.2.2

[9] 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

[10] 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/. 2.1

[11] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using Data Groups to Specify and Check Side Effects. In *Conference on Programming Language Design and Implementation*, 2002. ISBN 1-58113-463-0. doi: 10.1145/512529.512559. URL http://doi.acm.org/10.1145/512529.512559. 2.1

[12] 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. 2.1

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

[14] 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

[15] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006. 1

[16] 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

[17] 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

[18] 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

[19] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *Programming Languages and Systems*, 2009. ISBN 978-3-642-00590-9. 2.2

[20] 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. 2.1

[21] Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In *European Symposium on Programming Languages and Systems*, 1999. 2.1

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

[23] 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

[24] 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. 2.2