Controlling Module Authority Using Programming Language Design

Darya Melicher

CMU-ISR-20-101

February 2020

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
Lujo Bauer
Limin Jia
Alex Potanin, Victoria University of Wellington
Robert Biddle, Carleton University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

This research was supported by the National Security Agency (NSA) Lablet Contract No. H98230-14-C-0140 and the Defense Advanced Research Projects Agency (DARPA) agreement No. FA8750-16-2-0042. Any opinions, findings, and conclusions or recommendations expressed in this manuscript are those of the author(s) and do not necessarily reflect the views of NSA or DARPA. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Keywords: Language-based security, capabilities, authority, modules, effects

Abstract

The security of a software system relies on the principle of least privilege, which says that each software component must have only the privilege necessary for its execution and nothing else. Current programming languages do not provide adequate control over the privilege of untrusted software modules. To fill this gap, we designed and implemented a capability-based module system that facilitates controlling what resources each software module accesses. Then, we augmented our module system with an effect system that facilitates controlling how resources are used, i.e., authority over resources. Our approach simplifies the process of ensuring that a software system maintains the principle of least privilege. We implemented our solution as part of the Wyvern programming language.

In Wyvern, modules representing or using system resources, such as the file system and network, are considered to be security-critical and are designated as resource modules. References to resource modules are capability-protected, i.e., to access a resource module, the accessing module must have the appropriate capability. Using this feature, we designed our module system in such a way that it is obvious at compile time what capabilities a module have from looking at modules' interfaces and not their code. This property significantly simplifies the task of checking what capabilities a module holds. From a theoretical viewpoint, our capability analysis uses a novel, non-transitive notion of capabilities, which allows estimating the capabilities each module holds more precisely than in previous formal systems.

Further, leveraging the fact that effects are a good proxy for operations performed on a resource, we designed Wyvern's effect system that can account for the effects a module has on each resource. Our effect system is capability-based and allows specifying and enforcing what operations a module can perform on a resource it accesses, i.e., allows controlling the module's authority. Similarly to our module-system design, effect annotations that convey information about module authority are located in modules' interfaces, thus simplifying the task of checking resource usage.

We formalized both Wyvern's module system and effect system, and proved Wyvern to be capability- and authority-safe. We also assessed the effectiveness of the module system and the effect system that we designed in terms of how they would be used in practice and how they benefit a security-minded software developer writing an application. To do that, we implemented an extensible text-editor application in Wyvern and performed a security analysis on it.

Acknowledgments

As many pointed out before me, obtaining a doctorate degree is a journey, and everyone's journey is unique. My journey had a considerable effect on every aspect of my life, professional as well as personal. During my journey, I gained significant new knowledge, learned and practiced a variety of new skills, and met many people who affected my ultimate path.

I am tremendously grateful to my advisor Jonathan Aldrich for his guidance, help, and support, which were instrumental for me obtaining the doctoral degree. I am also indebted to my thesis committee for providing their useful advice and feedback. In addition, I would like to thank my collaborators, family, friends, fellow students, university and department staff, and various members of the research community.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The principle of least privilege [65] is a fundamental technique for designing secure software systems. It states that each component of a system must be able to access only the information and resources that it needs for operation and nothing more. For example, if an application module needs to append an entry to an application log, that module should not also be able to access the whole file system. This is important for any software system that divides its code into a trusted code base [63] and untrusted peripheral code because, in such a system, trusted code could run directly alongside untrusted code. A common example of such software systems is extensible applications that allow supplementing their functionality with third-party extensions (also called plugins, add-ins, and add-ons). Another example is large software systems in which some developers may lack the expertise to write secure or privacy-compliant code and thus should have a limited ability to access system resources in their code. Enforcing the principle of least privilege helps to limit the attack surface of a software system and to isolate vulnerabilities and faults. However, current programming languages do not provide adequate control over the authority of untrusted modules [12, 13, 39, 66, 70, 75]. For example, to control module privilege, Java uses sandboxing via the Java Security Manager, which is complicated to use, resulting in compromised protection of Java applications [13, 39].

Application security becomes even more challenging if an application uses code-loading facilities or advanced module systems, which allow modules to be dynamically loaded and manipulated at run time. In such cases, an application has extra implementation flexibility and may decide what modules to use at run time, e.g., responding to user configuration or the environment in which the application is run. On the other hand, untrusted modules may get access to crucial application modules that they do not explicitly import via global variables or method calls. For example, although an untrusted third-party extension may import only the logging module and not the file I/O module, the extension could still receive an instance of the file I/O module via a method call as an argument or a return value. Dynamic module loading can be modeled as first-class modules, i.e., modules that are treated like objects and can be instantiated, stored, passed as an argument, returned from a function, etc. However, in a conventional programming language featuring first-class modules (e.g., SML/NJ [30], Newspeak [8], Scala [54], and Grace [28]), because module passing can happen anywhere in an application's code, it is impossible to limit the amount of code one must inspect to check application security, making it difficult to track and control module accesses in practice.

Another challenge of ensuring an application's security is controlling the operations performed on the legitimately accessed resource. For example, in a text editor, a theme plugin may access the user-interface-related modules and may call methods that change the color of buttons, but it must not call methods that delete or disable buttons. So far, although there has been some success addressing the challenge of controlling module accesses (e.g., [41] and [16]), not much progress has been made towards controlling operations that are performed on modules.

1.1 Thesis Approach

Before discussing how the thesis approach described in this dissertation tackles the above-mentioned shortcomings of current programming languages, let us define the main terms:

- A **capability** is an unforgeable, communicable reference that allows accessing a resource. (In our approach, it is a reference to an object representing a resource.)
- A **capability-safe programming language (or system)** is a programming language (or system) in which capability passing is the main way for granting access rights, is restricted, and abides by a set of clearly defined rules.
- **Authority** is the ability to operate on resources. (In our approach, it is the ability to operate on objects representing resources.)
- An **authority-safe programming language (or system)** is a programming language (or system) that provides a way to specify and enforce a set of rules according to which authority over resources may be obtained.

To address the challenge of controlling modules' accesses, we developed a capability-safe module system that facilitates controlling the capabilities granted to each application module (Chapter 2). Unlike prior research, our capability analysis defines the capabilities possessed by an object non-transitively, allowing engineers to reason about programs that use wrappers to provide an attenuated version of a more powerful capability [48]. Furthermore, our approach simplifies reasoning about capabilities a module has. To determine a module's capabilities, software developers need to examine only the capabilities that are passed as module arguments when the module is created, or are delegated to the module later during execution. The type system facilitates this by both identifying which objects provide capabilities to system resources and enabling software developers to examine the capabilities passed into and out of a module based only on the module's interface, without needing to inspect the module's implementation.

To address the challenge of controlling modules' authority, i.e., controlling the operations modules perform on important, security- and privacy-crucial modules, we developed an authority-safe effect system that uses as a basis the capability-safe module system (Chapter 3). An effect system [37] is a formal system that keeps track of effects of a program, where an effect describes an action performed on a resource. Notably, the definition of an effect coincides with the information we are interested in from the security standpoint: resources are the security- and privacy-crucial modules of the program, and the actions performed on resources are the operations performed on the security- and privacy-crucial modules. For instance, using our effect system design, the programmer of the text-editor application from the example above is able to use effects to express the constraints that the plugin could only change the color of buttons but

not disable or delete them.

Strengthening the connection between effects and resources, in our effect system, effects are defined using object capabilities that represent resources. For example, implemented in our system, effects of the text-editor's theme plugin, along with a specification of all the operations performed on a button resource, contain an object capability (i.e., an object reference) to the button resource itself. This connection between resources and the operations performed on them closely and conveniently ties the two notions together aiding a software architect or a security analyst to reason about software system's architecture and security.

Since capability safety and authority safety require substantial support from the programming language, we designed a new programming language, called Wyvern [52], which is built from the ground up with the security goals in mind. Thus, both the capability-safe module system and effect system were developed as part of Wyvern, making it a capability- and authority-safe language.

1.2 Threat Model

Our approach helps software developers minimize the risks associated with handling potentially malicious third-party code that runs alongside the application code, such as a third-party library or application extension. Wyvern's module system and effect system allow software developers to specify and enforce security policies using module interfaces. In particular, to specify resource access and use, software developers, including authors of third-party code, must request resource-associated capabilities to be passed into module constructors and methods, and annotate module methods with appropriate effects.

We do not prescribe a single notion of authorized and unauthorized usage but instead provide the tools for software developers to establish boundaries between parts of the application code with varying levels of trust. The notion of authorized and unauthorized usage is defined by security analysts who audit the application code. For example, a security analyst may want to enforce that a third-party library is not allowed to access the network but can read from a specific file, and we aim to allow both the security analyst and the software developer (who may be the same person but act in different roles at different times) to specify and enforce this property at the module boundary level.

Using the static checks of Wyvern's type-and-effect system, we aim to prevent attacks that try to gain unauthorized privilege, and specifically, we consider the case when third-party code (e.g., an application extension) attempts to exploit the application either by:

1) gaining unauthorized capabilities (e.g., access to system resources or the privilege to load—malicious or vulnerable—native code) or
2) calling unauthorized methods on capability-protected resources (e.g., an application extension whose primary functionality requires read-only access to a file performs a write operation on that file).

In addition, there are several low-level attacks that are either not possible or out of scope for this work:

1. Attacks involving overflows, such as buffer overflow and integer overflow, are generally not possible in Wyvern, as all the languages used as Wyvern backends (currently Java,

JavaScript, and Python) are memory-safe.
2. Code injection is not possible in Wyvern, as Wyvern does not provide support for dynamic evaluation of expressions, such as an `eval` function.
3. Code injection in auxiliary languages, such as SQL queries, are explicitly out of scope for this project but can be mitigated using Wyvern's type-specific languages (TSLs) [55].
4. Attacks involving malicious or vulnerable native code (e.g., code that is vulnerable to code-injection attacks) are out of scope, as native libraries are able to bypass the compile-time enforcement inside Wyvern. However, these attacks can be mitigated by:
   (a) The (capability-based) control within Wyvern over what native code is loaded;
   (b) Keeping the loaded native code minimal;
   (c) Auditing the loaded native code;
   (d) For Java, compiling the Wyvern code that interoperates with the Java code into bytecode with appropriately set Java Security Manager restrictions, which is potential future work;
   (e) For JavaScript, compiling the Wyvern code that interoperates with Secure EcmaScript [2], which is potential future work.

1.2.1 Attack Scenarios

Our attack scenarios assume three actors:

1. A **software developer** writes application code, is not actively malicious, but is potentially fallible. The software developer may integrate a third-party library from a potentially malicious attacker.
2. A **security analyst** decides a security policy for the application and audits its code (written by software developers) as well as the resources and their use by third-party libraries. During an audit, the security analyst may modify code to enforce security policies.
3. An **attacker** is malicious and may provide both input to the application and potentially malicious third-party library code. The attacker attempts to take advantage of the application either indirectly by exploiting fallible code in the application or directly by gaining access to a system resource and using it, disobeying the policy set by the security analyst.

Notably, the **security analyst** and the **software developer** may be the same person, in that first, during development, the software developer or security analyst writes a security policy for enforcement (i.e., module interfaces) but, later, is writing implementation code and may make mistakes in implementation that they would like to be caught by Wyvern's type-and-effect system.

We are concerned with the attack scenario when a software developer is integrating a library provided by a potentially malicious third party. We aim to provide a mechanism to limit the resources given to such a library and provide an easy way for the software developer to check the resources that the library requests and how the requested resources are used. The software developer may change the application code, but not third-party code, to enforce security policies.

1.3 Thesis Statement and Outline

The thesis of this dissertation can be stated as follows:

> *A programming language can provide facilities to help a software developer in identifying and controlling the authority of software modules. Specifically, a software developer can leverage a capability-safe module system to control module accesses and an effect system to control how modules are used, i.e., modules' authority. It is feasible to combine the capability-safe module system with the effect system to yield an authority-safe programming language.*

Furthermore, the thesis statement can be broken down into the following four hypotheses:

Hypothesis 1: It is possible to design a capability-safe, first-class, higher-order module system.

Hypothesis 2: It is possible to design an effect system that allows controlling the authority each program module has.

Hypothesis 3: A capability-safe module system in conjunction with an effect system that allows controlling modules' authority can make a programming language authority safe.

Hypothesis 4: An authority-safe programming language can be used by software developers to control and reason about software modules' authority.

To examine these hypotheses, we developed and implemented in the Wyvern programming language the capability-safe module system (Chapter 2; **Hypothesis 1**) and the effect system (Chapter 3; **Hypothesis 2** and **Hypothesis 3**). To evaluate the effectiveness of the proposed programming-language designs, we developed and analyzed an extensible text-editor application (Chapter 4; **Hypothesis 4**).

Chapter 2

A Capability-Safe Module System

Wyvern modules have several features that distinguish our module system from others. First, modules are first-class, i.e., they are treated as objects and can be instantiated, stored, passed as arguments into methods, and returned from methods. Second, modules are treated as capabilities, i.e., unforgeable tokens that enable access to a module. Since modules are objects in Wyvern, modules act as object capabilities (objects in Wyvern act as object capabilities too). If a module can access another module, we say that the former module has a capability to use the latter module. Finally, modules are divided into two categories: *resource modules*, i.e., security- or privacy-related modules (system resources, modules containing application data, or state-bearing modules), and *pure modules*, i.e., non-state-bearing utility modules.

2.1 Running Example

To illustrate our approach, let us consider a sample application that allows third-party extensions. Figure 2.1 shows a module access diagram of a word-processor application, similar to OpenOffice or Microsoft Word, that extends its feature set by allowing third-party extensions. The vertical dotted line represents a virtual border between standard language-provided libraries and the word processor's code. Boxes represent modules, which are clustered according to their conceptual type. Boxes with a solid outline represent resource modules, and boxes with dotted outline represent pure modules. Arrows represent module accesses. If an arrow goes from module A to module B, module A accesses module B. Being able to access a resource module is equivalent to having a capability to access the imported module.

Wyvern provides a number of standard libraries: *Collections* refer to a set of pure modules that provide implementations of basic functionality, e.g., list and queue factories. *System Resources* refers to a set of language-provided modules that implement system-level functionality, e.g., file and network access. *Platforms* refer to the modules that implement the Wyvern back end. Platforms and system resources may be used to subvert the word-processor application or the machine it is running on, and thus access to them requires the possession of appropriate capabilities.

The word-processor system consists of core modules, which are considered trusted, and extension modules (marked so on the diagram), which are provided by third parties and considered

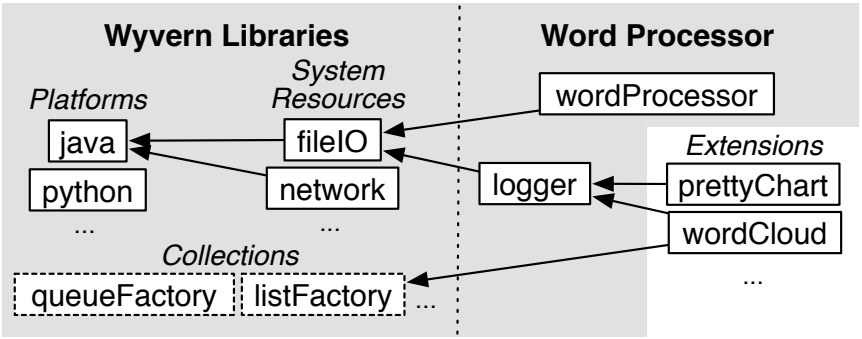


Figure 2.1: A module access diagram of the word-processor application used in code examples. Boxes represent modules. Boxes with a solid outline represent resource modules, and boxes with dotted outline represent pure modules. Arrows represent module accesses. If an arrow goes from module A to module B, A accesses B. The dark background delineates the trusted code base.

untrusted. The diagram presents only a subset of modules of the word processor's core that are used in our examples: the `wordProcessor` module is the main module of the word processor, and the `logger` module provides a logging service and can be used by multiple word processor's modules.

We use the word processor example to introduce Wyvern's two types of modules—resource modules and pure modules—and to show how one can determine a module's capabilities.

2.2 Resource Modules

Resource modules are defined as modules that encapsulate system resources (e.g., `java` and `fileIO`), use other resource modules (e.g., `wordProcessor` and `logger`), or contain mutable state (e.g., `wordProcessor`). A module is a resource if it has *at least one* of these characteristics. For example, the `wordProcessor` module is a resource module because it uses the system resource `fileIO` and has state (details upcoming). It is important for state-bearing modules to be resources, as they may contain private application data and also may facilitate communication between modules that access them, potentially allowing illegal sharing of capabilities.

Figure 2.2 presents a code example with several resource modules and types. By convention, in Wyvern, type names are capitalized, and module names (like variables, functions, and other identifiers that stand for values) start with lowercase letters. The code snippet starts with the abbreviated definition of the resource type `FileIO` that gives access to the file system, followed by the resource type `Logger` and a logging module of that type. For its operation, the `logger` module needs to use an instance of a module of type `FileIO`, and thus `logger` is a resource module. We prohibit global state and restrict access to resources, and so, to access a resource, a module must receive a reference to that resource. In Wyvern, this can be done via the argument-passing mechanism, and thus Wyvern's resource modules are ML-style *functors* [40]. Resource modules are functions that accept one or more arguments, each of which is a module instance of a required type, and produce a module instance as a result. In the case of `logger`, the module functor accepts a module instance of type `FileIO` and returns an instance of the `logger` module.

```
resource type FileIO
  def getStandardLogFile(): File
  ...

resource type Logger
  def appendToLog(entry: String): Unit

module def logger(io: FileIO): Logger
  def appendToLog(entry: String): Unit
    io.getStandardLogFile().append(entry)

module def wordProcessor(io: FileIO): WordProcessor
  import logger
  var log: Logger = logger(io)
  ...
```

Figure 2.2: A Wyvern code example demonstrating resource modules, how they are accessed, and their instantiations.

Following the `logger` module is the `wordProcessor` module, which is the main module of the word processor application. Since `wordProcessor` receives as an argument an instance of the `FileIO` type, `wordProcessor` is a resource module too. To access a resource module of the `FileIO` type, `wordProcessor` needs to have an appropriate capability. The capability must be passed into the `wordProcessor` module on its instantiation by either another module or top-level code.

The `wordProcessor` module instantiates the `logger` module by, first, importing the definition of the `logger` module using the `import` keyword and then calling the imported `logger` functor definition with appropriate arguments to get an instance of the `logger` module. The argument that `logger` requires is a module instance of the `FileIO` type, and by passing in `io`, `wordProcessor` gives `logger` the capability to use the module instance of the `FileIO` type it received on instantiation. The created instance of `logger` is immediately assigned to a local variable `log`, which may be used later in the `wordProcessor`'s code. Note that `wordProcessor` *receives* a module instance of the `FileIO` type as an argument, but it *instantiates*, i.e., creates a local instance of, the `logger` module. Generally, any resource module can instantiate other resource modules from its initialization block and even provide them with access to resource modules to which it itself has access. Since `logger` is a resource module, instantiating it creates a capability for it, which, in this case, belongs to the `wordProcessor` module.

Alternatively, if `wordProcessor` did not want to provide `logger` access to the file system, `wordProcessor` could create and pass in a dummy module of type `FileIO` as follows:

```
module def wordProcessor(io: FileIO): WordProcessor
  import logger
  var dio: FileIO = dummyIO
  var log: Logger = logger(dio)
  ...
```

This would disallow the `logger` module from having any access to the file system.

To run the program, the top-level code is as follows:

```
platform java
import fileIO
import wordProcessor
let io = fileIO(java) in
  let wp = wordProcessor(io) in ...
```

First, the back end to be used is specified using the `platform` keyword. This keyword can appear only on the top level and is used to create a resource module instance representing the back-end implementation. Then, the definitions of the `fileIO` and `wordProcessor` module functors are imported, and the two modules are instantiated receiving the arguments they require. The two newly created module instances are assigned to two variables in two nested `let` constructs and can be used in the rest of the code contained in the inner `let`'s body.

The top-level code exercises high-level control over accesses to resource modules, performing two important functions. First, it instantiates resource modules, implicitly creating capabilities that allow using the instantiated modules. Second, it grants module access capabilities (conceptually, in the Newspeak style [8]; syntactically, in the ML-functor style [40]): the instantiated modules (and implicit capabilities to use them) are passed as arguments to authorized modules.

For brevity, the top-level code can be shortened as follows:

```
require fileIO: FileIO
import wordProcessor
let wp = wordProcessor(fileIO) in ...
```

Here we use syntactic sugar (the keyword `require`) for specifying the platform (the default platform is chosen), importing the functor definition of the `fileIO` module, and instantiating it. This syntactic sugar can be used for resource modules that require only the back-end platform implementation in any programs that benefit from abstracting the platform they run on.

2.3 Pure Modules

The definition of a pure module is the opposite of the definition of a resource module. Pure modules are those modules that do not encompass system resources, do not access any resource module instances, and do not capture or transitively reference any mutable state. For a module to be pure, *all* of these conditions must be satisfied. The last condition has a caveat. The prohibition is on whether a module and its functions *capture* state, not whether they *affect* it. Functions defined in a pure module may have side effects on state, but only if the state in question is passed in as an argument or created within the function itself. Thus, since pure modules do not add to the caller's ability to use resources or cause side effects, pure modules are harmless from the security perspective. For more convenience, in Wyvern, any module can import any pure module.

Figure 2.3 shows an example of a pure module and how it can be imported. The `listFactory` module is the implementation of a list factory and belongs to the standard Wyvern library. It does not contain mutable state, but only creates new lists, and therefore is a pure module. In Wyvern, pure modules are *not* functors, and a module that imports a pure module receives an instance of the pure module.

```
module listFactory: ListFactory
  def create(): List
    ...

module def wordCloud(log: Logger): WordCloud
  import listFactory as list
  var words: List = list.create()
  ...
```

Figure 2.3: A Wyvern code example demonstrating a pure module and its import.

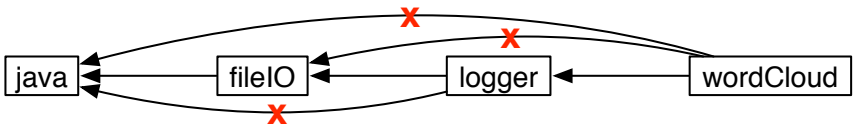


Figure 2.4: Access capabilities of `fileIO`, `logger`, and `wordCloud`. If an arrow goes from module A to module B, A has a capability to access B. Crosses on arrows mean that such access capability is not granted. In Wyvern, capabilities are non-transitive.

The `wordCloud` module is a third-party extension module that creates a word cloud—an image composed of words used in a text passage, in which the size of each word indicates its frequency—and pastes it into a word processor document. The `wordCloud` module uses a list to store the words it operates on and therefore imports the `listFactory` module using the `import` keyword. Since, for pure modules, the import statement produces a module instance, it can be immediately assigned to a local variable using the `as` keyword. The import of `listFactory` by `wordCloud` is invisible to the module or top-level code that instantiates the `wordCloud` module.

2.4 Capability Analysis

As stated in our threat model, we are concerned with the capabilities granted to third-party extensions, as well as minimizing access to system resources by all application modules. In this section, we demonstrate how a security analyst can verify that capabilities of the modules in the word-processor application match the capabilities shown in Figure 2.4. (In Section 2.6, we show how to determine capabilities of arbitrary objects and provide a formal definition.)

Since access to resources is mediated by modules, we can represent the capabilities of a given module as the set of resource modules it can access. In Figure 2.4, if an arrow goes from module A to module B, A has a capability to access B. If an arrow is crossed, it means that such access capability is not granted. Thus, `wordCloud` has a capability to access `logger`, which in turn has a capability to access `fileIO`, which ultimately has access to the `java` foreign function interface module. We want to verify that the transitive extension of these capabilities relationships does not hold, e.g., the `wordCloud` module does not have a capability to access the file system or perform file I/O operations supported by the `fileIO` module. In effect, we are verifying that `wordCloud` gets only an attenuated capability to do file I/O, i.e., `wordCloud` can perform the logging oper-

ations supported by the `logger` module and nothing more. This facilitates a defense-in-depth strategy: if an attacker controls the `wordCloud` module and somehow subverts the `logger` module to get a `fileIO` capability, since `fileIO` itself attenuates the `java` foreign function interface capability, the attacker can do file I/O but cannot make arbitrary system calls supported by the Java standard library.

To verify that capabilities are property attenuated (thereby mitigating the attack mentioned above by ensuring that `wordCloud` cannot get a `fileIO` capability), we need to check that the `fileIO` module is properly encapsulated by the `logger` module, and that the `logger` module provides operations that are restricted appropriately to the intended semantics of logging and cannot be used to do arbitrary file I/O.

We can check encapsulation by manually inspecting the interface of `wordCloud` as well as the interfaces of the modules it accesses: `Logger` and `ListFactory`. Since `ListFactory` is not a resource module, we do not have to look any further at its interface. Note that, in contrast to dynamically typed, capability-safe languages such as E [47] or Newspeak [8], Wyvern's type system aids our inspection here. We inspect the interface of `logger` and immediately observe that none of the types in `logger`'s interface are resource types. Thus, we verify that `logger` cannot leak a reference to the `fileIO` module that it uses internally—again, using only the type of the `logger` module, not its implementation.

Of course, encapsulation by itself is not enough: if `logger` provided the same operations as `fileIO`, it would essentially provide the same capabilities despite the actual `fileIO` being encapsulated. To this end, we check that `logger` attenuates the capability to access `fileIO` and that `logger` can only do logging, instead of arbitrary file operations, by looking at the implementation of `logger`. Notably, this manual inspection is localized: we can use interfaces to reason about where capabilities can reach and then check the code that uses those capabilities to ensure it enforces the proper invariants. We do not have to inspect any code if we can show that the capability we are reasoning about does not reach that code. In this case, if we do inspect `logger`, it is easy to see that it invokes `append` on a specific file, which is characteristic of the intended logging functionality.

This process would be more complicated in a language that is not capability-safe or even in a language that is capability-safe but does not have Wyvern's static typing support. In a language that is not statically typed, we could not so quickly exclude the possibility that a capability of interest is hidden in `ListFactory`, nor could we be sure that we know all of the operations available on an object unless we enforce that dynamically by imposing a wrapper. In a language that is not capability-safe, there is much more to worry about: `wordCloud` could get access to `fileIO` by reading a global variable, a reference to a file object could be smuggled in an apparently innocent variable of type `Object` and then downcast to type `File`, or reflection could be used to extract a `fileIO` reference from within the `logger` object. However, these are not possible in Wyvern: currently, Wyvern does not support arbitrary downcasts but only pattern matching in a hierarchy where the possible child types are known. In addition, Wyvern's capability-safe reflection mechanism respects type restrictions [74], so that reflection cannot be used to do anything other than invoke the public methods of `logger`. Thus, Wyvern's capability-safe module system along with its static types greatly simplify reasoning about modules' capabilities.

$$
\begin{array}{rcll}
p & ::= & \overline{md}\ \texttt{platform}\ x\ \ \overline{i}\ e & \textit{program} \\
md & ::= & h\ \overline{i}\ \overline{d} & \textit{module} \\
h & ::= & \texttt{module}\ x : \tau & \textit{module header} \\
 & | & \texttt{module def}\ x(\overline{y : \tau}) : \tau & \\
i & ::= & \texttt{import}\ x\ [\texttt{as}\ y] & \textit{imports} \\
d & ::= & \texttt{def}\ m(\overline{x : \tau}) : \tau = e & \textit{declarations} \\
 & | & \texttt{var}\ f : \tau = x &
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & x & \textit{expressions} \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) & \\
 & | & e.m(e) & \\
 & | & e.f & \\
 & | & e.f = e & \\
 & | & \texttt{let}\ x = e\ \texttt{in}\ e & \\
 & | & \texttt{bind}\ \overline{x = e}\ \texttt{in}\ e & \\
s & ::= & \texttt{resource} \mid \texttt{pure} &
\end{array}
$$

Figure 2.5: Wyvern's abstract grammar.

2.5 Formalization

Although modules are at the heart of our work, they are not central to Wyvern's formal system. Inspired by the Wyvern core work [52], our modules are syntactic sugar on top of an object-oriented core language and are available for software developers' convenience. We present the Wyvern formal system in the following order: first, we describe the abstract grammar for writing modules in Wyvern, then the object-oriented core language syntax and module translation into it, and finally, Wyvern's static and dynamic semantics. This precisely defines our design and lays the groundwork for the definition and proof of capability safety in Section 2.6.

2.5.1 Module Syntax

Wyvern's abstract grammar is shown in Figure 2.5. A Wyvern program consists of zero or more modules followed by the top-level code that includes specifying the back end used to run the program using the `platform` keyword, zero or more module imports, and an expression e. Each module consists of a module header h, a list of imports $\overline{i}$, and a list of declarations $\overline{d}$. Module headers can be one of two types depending on whether the module is a resource module or a pure module. If a module is pure, its header consists of the `module` keyword, a name x that uniquely identifies the module, and a module type τ. If a module is a resource module, its header consists of the `module` keyword, followed by the `def` keyword, which signifies that it is a functor, a name x, which uniquely identifies the module functor, a list of functor parameters and their types, and a functor return type τ.

The module-import syntax is used for importing instances of pure modules or module functors for resource modules, and consists of the `import` keyword followed by the module or functor name x. In the case of importing an instance of a pure module, for convenience, the instance can be renamed using the `as` keyword.

A module can contain declarations of two kinds: method declarations and variable declarations. Method declarations are specified using the `def` keyword followed by the method name m, a list of method parameters and their types, the method's return type τ, and the method body e. Variable declarations are specified using the keyword `var` followed by the variable name f, the variable type τ, and the value x. We restrict the form of the initialization expression to simplify

$$
\begin{array}{rcll}
e & ::= & x & \textit{expressions} \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) & \\
 & | & e.m(e) & \\
 & | & e.f & \\
 & | & e.f = e & \\
 & | & \texttt{bind}\ x = e\ \texttt{in}\ e & \\
 & | & l & \\
 & | & l.m(l) \rhd e & \\
s & ::= & \texttt{resource} \mid \texttt{pure} & \\
d & ::= & \texttt{def}\ m(x : \tau) : \tau = e & \textit{declarations} \\
 & | & \texttt{var}\ f : \tau = x & \\
 & | & \texttt{var}\ f : \tau = l & \\
\tau & ::= & \{\overline{\sigma}\}_s & \textit{object type}
\end{array}
\qquad
\begin{array}{rcll}
\sigma & ::= & \texttt{def}\ m(x : \tau) : \tau & \textit{declaration types} \\
 & | & \texttt{var}\ f : \tau & \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x : \tau & \textit{var. typing context} \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s & \textit{store} \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l : \tau & \textit{store typing context} \\
E & ::= & [\,] & \textit{evaluation context} \\
 & | & E.m(e) & \\
 & | & l.m(E) & \\
 & | & E.f & \\
 & | & E.f = e & \\
 & | & l.f = E & \\
 & | & \texttt{bind}\ x = E\ \texttt{in}\ e & \\
 & | & l.m(l) \rhd E &
\end{array}
$$

Figure 2.6: Syntax of Wyvern's object-oriented core.

translation into the core, but this is relaxed in our implementation.

Wyvern expressions are common for an object-oriented programming language and include: a variable, the `new` construct, a method call, a field access, a field assignment, and the `let` and `bind` constructs. The `new` construct carries a tag s that indicates whether the object being created is pure or is a resource, which is at the core of our formalization of capability control. It also contains a self reference x that is similar to a `this`, but provides more flexible naming, and is used for tracking the receiver (discussed in more detail later). In our implementation, x defaults to "this" when no name is specified by the programmer. Finally, the `new` construct accepts a list of declarations $\overline{d}$. The `bind` construct is similar to a `let` with the difference that expressions in its body can access only the variables defined in it and nothing outside it (one can think of it as a Scala's spore [46] or an AmbientTalk's isolate [72]). The types of variables defined in a `let` or `bind` are inferred.

2.5.2 Core Syntax

For the sake of uniformity and to simplify reasoning about capability safety, Wyvern modules are translated into objects. The abstract grammar that has modules (Figure 2.5) is translated into the object-oriented core of Wyvern that does not have modules (Figure 2.6). Furthermore, in Wyvern's object-oriented core:

- Methods may have only one parameter.
- Expressions do not include the `let` construct.
- The `bind` construct may have only one variable.
- Expressions and declarations are extended with run-time forms that cannot appear in the source code of a Wyvern program.

To represent multiparameter methods, the `let` construct, and multivariable `bind` in the object-oriented core, we use a standard encoding (presented in the next section).

Expressions have two run-time forms: a location and a method-call stack frame. The location l refers to a location in the store μ (on the heap) that holds an object definition added at

$$
\begin{aligned}
trans(\overline{md}\ \texttt{platform}\ z\ \ \overline{i}\ e) &= \begin{cases} \texttt{let}\ name(md) = trans(md) & \text{if } \overline{md} = md\ \ \overline{md'} \\ \texttt{in}\ trans(\overline{md'}\ \texttt{platform}\ z\ \ \overline{i}\ e) & \\ \texttt{bind}\ z = \langle constResObj \rangle\ trans(\overline{i}) & \text{if } \overline{md} = \varnothing \\ \texttt{in}\ e & \end{cases} \\
trans(\texttt{module}\ x : \tau\ \ \overline{i}\ \ \overline{d}) &= \texttt{bind}\ trans(\overline{i})\ \texttt{in}\ \texttt{new}_{\texttt{pure}}(x \Rightarrow \overline{d}) \\
trans(\texttt{module def}\ x(\overline{y : \tau}) : \tau\ \ \overline{i}\ \ \overline{d}) &= \texttt{new}_{\texttt{resource}}(x \Rightarrow \texttt{def}\ apply(\overline{y : \tau}) : \tau \\
&\qquad \texttt{bind}\ \overline{y = y}\ trans(\overline{i}) \\
&\qquad \texttt{in}\ \texttt{new}_{\texttt{resource}}(\_ \Rightarrow \overline{d})) \\
trans(\overline{i}) &= \begin{cases} y = x\ \ trans(\overline{i'}) & \text{if } \overline{i} = \texttt{import}\ x\ \texttt{as}\ y\ \ \overline{i'} \\ \varnothing & \text{if } \overline{i} = \varnothing \end{cases} \\
name(\texttt{module}\ x : \tau\ \ \overline{i}\ \ \overline{d}) &= x \\
name(\texttt{module def}\ x(\overline{y : \tau}) : \tau\ \ \overline{i}\ \ \overline{d}) &= x
\end{aligned}
$$

$$
\begin{aligned}
\texttt{let}\ x = e\ \texttt{in}\ e' &\equiv \texttt{new}_s(\_ \Rightarrow \texttt{def}\ f(x : \tau) : \tau' = e').f(e) \\
\texttt{bind}\ \overline{x = e}\ \texttt{in}\ e &\equiv \texttt{bind}\ x = (e_1, e_2, ..., e_n)\ \texttt{in}\ [x.n/x_n]e \\
\texttt{def}\ m(\overline{x : \tau}) : \tau = e &\equiv \texttt{def}\ m(x : (\tau_1 \times \tau_2 \times ... \times \tau_n)) : \tau = [x.n/x_n]e
\end{aligned}
$$

Figure 2.7: Modules-to-objects translation rules, and encodings for `let`, multivariable `bind` and multiparameter methods.

object creation. The method-call stack frame models the call stack and method calls on it, while preserving information about the receiver of the executing method. The expression $l.m(l_1) \triangleright e$ means that we are currently executing the method body e of a method m of the receiver l, and object l_1 was passed as an argument.

Declarations have only one run-time form for object fields. Method bodies can never contain method-call stack frames. An object field in the source code can contain only a variable, which at run time becomes a location in the store. Thus, the run-time form for an object field represents that a field f is referring to a location l.

A set of types of object fields and methods forms an object type, which is tagged as either pure or resource. We use standard typing contexts Γ for variables and Σ for the store, and to simplify Wyvern dynamic semantics, an evaluation context E.

2.5.3 Modules-to-Objects Translation

Figure 2.7 presents modules-to-objects translation rules and encodings that are used in the translation but not expanded for brevity. Overall, a Wyvern program is translated into a sequence of `let` statements, where every variable in a `let` represents a module and the body of the last `let` in the sequence is a `bind` expression containing the top-level code. The reason for this program structure is that module definitions are pure and thus can be created in the `let` statements with no restrictions. However, when modules are instantiated in the top-level code or inside other modules, they operate in the restricted environment of the `bind` statement in the body of the last `let`, having access only to the variables defined in that `bind`.

Variable names in the `let` expressions correspond to module names. Variables defined in the

`bind` expression that immediately surrounds the top-level code are a special constant resource object, representing the back-end implementation, and the translation of top-level imports.

In essence, modules are translated into objects: pure modules are translated into pure objects and resource modules are translated into resource objects. The exact translation of a Wyvern module depends on whether the module is a pure module or a resource module, but for both kinds of modules the object representing the module and containing all of the module's declarations is created in a restricted environment of the body of a `bind` expression. This ensures that, when a module is created, the module's declarations do not gain unauthorized access to resources outside those defined in the `bind`.

If the module is pure, it translates into a `bind` construct, in which the module's imports become the `bind`'s variables, and the module's declarations are wrapped into a pure object of type τ in the `bind`'s body. If the module is a resource module, it is a functor, and it translates into a new resource object with a single method `apply`. The `apply` method takes as arguments the functor's arguments and, when called, returns a `bind` expression. The variables in the returned `bind` consist of variables that shadow the functor's arguments (since a `bind`'s body can access only the variables defined in the `bind` and no other, outside variables) and the imports of the resource module under translation. The body of the `bind` contains a resource object that encompasses the declarations of the translated resource module. The module's declarations are prohibited from referring to the resource object itself (as it does not exist in the original code), and therefore we generate a fresh name for the self variable (in the translation, it is marked with an underscore). The `apply` method of a functor's translation is invoked whenever the functor is invoked.

Importantly, the `bind` construct plays a significant role in Wyvern's module access control. Module imports and arguments are translated into variables in a `bind` construct. Since the body of a `bind` is forbidden to access anything outside the variables defined in the `bind`, a module can receive a capability to access a resource only via the argument-passing mechanism of its functor, as an argument to one of its methods, or as the return value from a method call on an imported module. This substantially limits the number of possible paths for acquiring module access.

The `let` construct, a multivariable `bind` construct, and multiparameter methods are provided only for software developer convenience and are absent from Wyvern's core syntax; they are encoded instead. The `let` construct is encoded as a method call, and the multiplicity of variables in the `bind` construct and parameters in methods is achieved by bundling variables and parameters together in a tuple and then accessing them by their indices in the `bind` and methods' bodies.

Figure 2.8 shows an example of applying the translation rules from Figure 2.7. On the left is a code snippet as a developer would write it, and on the right is the same code written in Wyvern's core syntax without modules (the encodings are not expanded for conciseness, and we use the type abbreviations supported by our implementation rather than the less-readable structural types in our formalism). The snippet is a partial program; the `logger` and `fileIO` modules are assumed to be defined elsewhere.

The `listFactory` and `wordProcessor` modules are translated into variables defined in two nested `let`s. The outer `let` defines the `listFactory` module, which is translated into a `bind` expression. Since `listFactory` does not import any modules, the `bind` has no variables, and the `bind`'s body is a new pure object encompassing the `listFactory`'s `create` method. Being in the body of the `bind` expression and having no variables defined in the `bind`, `listFactory` has access to no resources and no other modules.

```
module listFactory: ListFactory
  def create(): List
    ...

module def wordProcessor(io: FileIO)
                      : WordProcessor
  import wyvern: listFactory as list
  import logger
  var log: Logger = logger(io)
  var exts: List = list.create()
  ...

// top level
platform java
import fileIO
import wordProcessor
let io = fileIO(java) in
  let wp = wordProcessor(io) in ...
```

```
let listFactory = bind in new_pure(x ⇒
  def create() : List = ...)
  in let wordProcessor = new_resource(x ⇒
    def apply(io : FileIO) : WordProcessor
      bind
        io = io
        list = listFactory
        logger = logger
      in new_resource(_ ⇒
        var log : Logger = logger.apply(io)
        var exts : List = list.create()
        ...))
// top level
in bind
  java = ⟨constResObj⟩
  fileIO = fileIO
  wordProcessor = wordProcessor
in
  let io = fileIO.apply(java) in
    let wp = wordProcessor.apply(io) in ...
```

Figure 2.8: A sample modules-to-objects translation. On the left is a code snippet as a developer would write it, and on the right is the same code written in Wyvern's core syntax without modules. The encodings are not expanded for conciseness, and we use the type abbreviations supported by our implementation rather than the less-readable structural types from the formalism. The `logger` and `fileIO` modules are assumed to be defined elsewhere.

The inner `let` defines the `wordProcessor` module, which is translated into a resource object containing an `apply` method. Similarly to the `wordProcessor` functor, the `apply` method takes an object of the `FileIO` type and returns an object of the `WordProcessor` type. The body of the `apply` method is a `bind` expression, the variables of which are the `apply`'s argument `io` as well as the two `wordProcessor`'s imports, `listFactory` and `logger`. The body of the `bind` expression has a resource object encompassing `wordProcessor`'s declarations. To get an instance of the `logger` module, the `logger`'s `apply` method is called on it with an appropriate argument. Since the body of the `bind` is limited to access only the variables defined in the `bind`, `wordProcessor` has access to only three modules, `fileIO`, `listFactory`, and `logger`, and no other modules.

The top-level code is translated as the body of the inner `let` and is represented by a `bind` expression. The `bind` expression has all top-level imports as variable definitions, and the `bind`'s body contains the rest of the top-level code (e.g., the two nested `let` expressions).

2.5.4 Static Semantics

The Wyvern static semantics is presented in Figure 2.9. The annotation on top of the turnstile represents the current or future (in case of object creation) receiver of the enclosing method.

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

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash^{e} x : \tau}\ \text{(T-VAR)} \qquad \frac{\Gamma,\ x : \{\overline{\sigma}\}_s \mid \Sigma \vdash_s^x \overline{d} : \overline{\sigma}}{\Gamma \mid \Sigma \vdash^{e} \texttt{new}_s(x \Rightarrow \overline{d}) : \{\overline{\sigma}\}_s}\ \text{(T-NEW)} \qquad \frac{\Gamma \mid \Sigma \vdash^{e'} e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \mid \Sigma \vdash^{e'} e : \tau_2}\ \text{(T-SUB)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e_1 : \{\overline{\sigma}\}_s \quad \texttt{def}\ \ m(x : \tau_2) : \tau_1 \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e} e_2 : \tau_2}{\Gamma \mid \Sigma \vdash^{e} e_1.m(e_2) : \tau_1}\ \text{(T-METHOD)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e : \{\overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma}}{\Gamma \mid \Sigma \vdash^{e} e.f : \tau}\ \text{(T-FIELD)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e_1} e_1 : \{\overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e_1} e_2 : \tau}{\Gamma \mid \Sigma \vdash^{e_1} e_1.f = e_2 : \tau}\ \text{(T-ASSIGN)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e_1 : \tau_1 \quad x : \tau_1 \mid \Sigma \vdash^{e} e_2 : \tau_2}{\Gamma \mid \Sigma \vdash^{e} \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{(T-BIND)} \qquad \frac{l : \tau \in \Sigma}{\Gamma \mid \Sigma \vdash^{e} l : \tau}\ \text{(T-LOC)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e'} l_1 : \{\overline{\sigma}\}_s \quad \texttt{def}\ \ m(x : \tau_2) : \tau_1 \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e'} l_2 : \tau_2 \quad \Gamma \mid \Sigma \vdash^{l_1} e : \tau_1}{\Gamma \mid \Sigma \vdash^{e'} l_1.m(l_2) \rhd e : \tau_1}\ \text{(T-STACKFRAME)}$$

$\boxed{\Gamma \mid \Sigma \vdash_s^z \overline{d} : \overline{\sigma}}$ $\boxed{\Gamma \mid \Sigma \vdash_s^z d : \sigma}$

$$\frac{\forall j,\ d_j \in \overline{d},\ \sigma_j \in \overline{\sigma},\ \Gamma \mid \Sigma \vdash_s^z d_j : \sigma_j}{\Gamma \mid \Sigma \vdash_s^z \overline{d}\ :\ \overline{\sigma}}\ \text{(T-DECLS)}$$

$$\frac{\begin{array}{c}\Gamma_{\texttt{resource}} = \{x : \{\overline{\sigma}\}_{\texttt{resource}} \mid x : \{\overline{\sigma}\}_{\texttt{resource}} \in \Gamma\} \\ \Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}} \qquad \Gamma_{\texttt{pure}},\ y : \tau_1 \mid \Sigma \vdash^z e : \tau_2\end{array}}{\Gamma \mid \Sigma \vdash_{\texttt{pure}}^z \texttt{def}\ m(y : \tau_1) : \tau_2 = e\ :\ \texttt{def}\ m(y : \tau_1) : \tau_2}\ \text{(DT-DEFPURE)}$$

$$\frac{\Gamma,\ x : \tau_1 \mid \Sigma \vdash^z e : \tau_2}{\Gamma \mid \Sigma \vdash_{\texttt{resource}}^z \texttt{def}\ m(x : \tau_1) : \tau_2 = e\ :\ \texttt{def}\ m(x : \tau_1) : \tau_2}\ \text{(DT-DEFRESOURCE)}$$

$$\frac{\Gamma \mid \Sigma \vdash^z x : \tau}{\Gamma \mid \Sigma \vdash_{\texttt{resource}}^z \texttt{var}\ f : \tau = x\ :\ \texttt{var}\ f : \tau}\ \text{(DT-VARX)}$$

$$\frac{\Gamma \mid \Sigma \vdash^z l : \tau}{\Gamma \mid \Sigma \vdash_{\texttt{resource}}^z \texttt{var}\ f : \tau = l\ :\ \texttt{var}\ f : \tau}\ \text{(DT-VARL)}$$

$\boxed{\mu : \Sigma}$

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

Figure 2.9: Wyvern static semantics.

Tracking the receiver is used in lieu of making object fields private. Both mechanisms enforce non-transitivity of capabilities, but receiver tracking is simpler and is already implemented for capability safety. The annotation underneath the turnstile—in the premise of T-NEW and dec-

laration typing rules—is the same as the tag on the `new` construct in the syntax and serves to identify objects and their declarations as `pure` or `resource`.

For expressions, the judgement reads that, in the variable typing context Γ and the store typing context Σ, with the receiver e' of the enclosing method, the expression e is a well-typed expression with the type τ. Similarly, for declarations, the judgement reads that, in the variable typing context Γ and the store typing context Σ, with the receiver z of the enclosing method, the declaration d that belongs to a pure or a resource object (the s tag underneath the turnstile) is a well-typed declaration with the type σ. The judgement for a set of declarations is analogous.

In the premise of the T-NEW rule, the receiver for the new object's declarations is the new object itself. In the conclusion of T-FIELD and T-ASSIGN, the receiver must be the object whose field is being accessed, which makes object field accesses private to the object to which they belong. For all declaration typing rules, the receiver is the object to which the declarations belong.

The T-DECLS rule enforces that each declaration of an object is well typed. DT-DEFPURE and DT-DEFRESOURCE typecheck pure and resource object methods respectively. A pure method should be able to typecheck in a typing environment without any resource variables, except for the passed argument. The argument may be a resource, but because all other variables in the context are pure, it cannot be stored (e.g., be assigned to a variable) inside the method body. If all methods in an object are pure and the object does not have any fields, the object is pure. DT-DEFRESOURCE has a standard, much less restrictive premise than DT-DEFPURE. If an object has a field, it is automatically declared a resource, and its typechecking proceeds as expected depending only on whether the field's value is a variable (DT-VARX) or a location (DT-VARL). The T-STORE rule ensures that the store is well formed.

To summarize, an object is a resource if at least one of the following conditions is true:

- The object contains a field (e.g., the object representing the `wordProcessor` module).
- An object's method definition needs a resource variable to typecheck (e.g., the object representing `logger` needs an object of type `FileIO` to typecheck).

These conditions are checked *statically*. If neither of them are true, then the object is pure (e.g., the object representing the `listFactory` module).

2.5.5 Subtyping Rules

Subtyping rules are presented in Figure 2.10. All of them are standard, except for the S-RESOURCE rule, which is used for the conversion between resource objects and pure objects. A pure object is a subtype of a resource object and, thus, can be used in place of a resource object but not the other way around.

2.5.6 Dynamic Semantics

Figure 2.11 shows Wyvern's dynamic semantics. The judgement is fairly standard and reads that, given the store μ, the expression e evaluates to the expression e' and the store becomes μ'.

The E-CONGRUENCE rule subsumes all evaluation rules with non-terminal forms; the rest of the reduction rules deal with terminal forms. To create a new object, a fresh store location is

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

$$\frac{}{\tau <: \tau} \text{ (S-REFL1)} \qquad \frac{\{\sigma_j^{j\in 1..n}\}_s \textit{ is a permutation of } \{\sigma_j'^{j\in 1..n}\}_s \quad n \geq 0}{\{\sigma_j^{j\in 1..n}\}_s <: \{\sigma_j'^{j\in 1..n}\}_s} \text{ (S-PERM)}$$

$$\frac{n, k \geq 0}{\{\sigma_j^{j\in 1..n+k}\}_s <: \{\sigma_j^{j\in 1..n}\}_s} \text{ (S-WIDTH)} \qquad \frac{\forall j,\ \sigma_j <: \sigma_j' \quad n \geq 0}{\{\sigma_j^{j\in 1..n}\}_s <: \{\sigma_j'^{j\in 1..n}\}_s} \text{ (S-DEPTH)}$$

$$\frac{}{\{\overline{\sigma}\}_{\texttt{pure}} <: \{\overline{\sigma}\}_{\texttt{resource}}} \text{ (S-RESOURCE)}$$

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

$$\frac{}{\sigma <: \sigma} \text{ (S-REFL2)} \qquad \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\texttt{def}\ m(x : \tau_1) : \tau_2 <: \texttt{def}\ m(x : \tau_1') : \tau_2'} \text{ (S-DEF)}$$

Figure 2.10: Wyvern subtyping rules.

chosen, and the object definition is assigned to it (E-NEW). In E-METHOD, when the method argument is reduced to a location, a method-call stack frame is put onto the stack, the caller and the argument are substituted with corresponding locations in the method body, and the method body starts to execute. An object field is evaluated to the location that it holds (E-FIELD), and when an object field's value is reassigned, the necessary substitutions are made in the store (E-ASSIGN). Similarly to methods, when the `bind`'s variable value is fully evaluated, variables in its body are substituted with their corresponding locations, and the `bind`'s body starts to execute (E-BIND). Finally, in the E-STACKFRAME rule, when a method body is fully executed, the method-call stack frame is popped from the stack and the resulting location is returned.

Notably, pure objects always remain pure, i.e., if a location l maps to a pure object in the store μ, then it always maps to a pure object in the store μ'. This can be proven by a simple induction on the reduction rules.

2.5.7 Type Soundness

The preservation and progress theorems are stated as follows. The proofs for both the theorems are fairly standard and are available in Appendix A.1.

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

Theorem (Progress)**.** *If* $\varnothing \mid \Sigma \vdash^{e''} e : \tau$ *(i.e.,* e *is a closed, well-typed expression), then either* e *is a value (i.e., a location), or* $\forall \mu$ *such that* $\mu : \Sigma$, $\exists e', \mu'$ *such that* $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$.

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

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

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

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

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

$$\frac{}{\langle \mathtt{bind}\ x = l\ \mathtt{in}\ e \mid \mu \rangle \longrightarrow \langle [l/x]e \mid \mu \rangle}\ \text{(E-BIND)} \qquad \frac{}{\langle l.m(l_1) \rhd l_2 \mid \mu \rangle \longrightarrow \langle l_2 \mid \mu \rangle}\ \text{(E-STACKFRAME)}$$

Figure 2.11: Wyvern dynamic semantics.

2.6 Capability Safety

We use the object-oriented core to prove our language capability-safe. Once modules are translated into objects, objects become the unit of reasoning, and thus our capabilities-related formalism is formulated in terms of objects.

In our system, a *principal* [15] is a resource object. An object—a principal or a pure object—can *directly access* a principal if the object has a reference to the principal, either by capturing it on object creation or acquiring it via a method call or return. The *capabilities* of an entity (an object or an expression) is the set of principals the entity can directly access.

The *capability-safety* property (formally defined in Section 2.6.2 below) states that the capabilities of an object can only increase due to the creation of a new object, a method call, or a method return. More precisely, the situations in which an object's capabilities can increase are:

1. **Object creation:** If a resource object A creates a new resource object B, then A has a capability to access B.
2. **Method call:** If a resource object A does not have a capability to access a resource object B and receives B as an argument to one of A's methods, then A acquires a capability to access B (perhaps only temporarily, while A's method is being executed).
3. **Method return:** If a resource object A does not have a capability to access a resource object B and B is returned from a method call that A invoked, then A acquires a capability B (perhaps only temporarily, while A's method is being executed).

It is important to note that these must be *the only* situations when the set of capabilities of an object increases (e.g., capabilities an object has cannot increase due to side effects). The capability-safety property is what assures us that all we need to reason about the capabilities of an object is to examine actions at its interface: method calls and returns; the case of object

creation is usually not very interesting because the newly created object is born with no more capabilities than its creator had.

Note that the third case of capability safety is unique to our non-transitive definition of capabilities. In the transitive definitions of capabilities used in prior work, the caller of a method always already has the same capabilities as its callee, or more. This also means that if an object such as the `logger` is careful not to return a reference to the underlying file being used, then objects that use the `logger` will not have a capability to access that file, which matches our intuition about the role of the `logger` object as a gatekeeper.

For a pure object, an increase in the set of its capabilities is inconsequential because a pure object cannot store mutable state. Thus, the definition of capability safety focuses on principals—i.e., resource objects. On a technical level—as discussed in more detail below—we treat a pure object as being part of whatever resource object uses it.

2.6.1 Significance of Capability Safety

If a Wyvern program typechecks, it is capability safe, i.e., capability gains are possible *only* in the three cases specified by the capability safety theorem. The type system *automatically, at compile time* enforces that a module *cannot* gain capabilities to access another module by any other means (e.g., via side effects). This property allows developers to reason effectively about the capabilities of program modules.

Consider reasoning about the capabilities of the `wordCloud` module. `wordCloud` is born with only the capability to access its required resources: due to the typechecking rule for `bind` and the way that modules are translated, these are the only resources in scope when `wordCloud` is instantiated. To see whether `wordCloud` acquires any additional capabilities, the capability-safety theorem tells us we need only inspect its type (`WordCloud`) and that of its required resources (`Logger`). Together the types show to what resources `wordCloud` can acquire capabilities via method calls and returns (cases 2 and 3 of the capability-safety theorem). For example, it is easy to verify that no object representing `fileIO` can go across this interface and thus ensure that all file access done by `wordCloud` must go through the `logger`. Case 1 of capability safety allows `wordCloud` to create objects of its own that act as principals, but it cannot thereby gain access to system resources it did not already have. Notice that we can conclude all of this without even looking at the code in the `wordCloud` module—which is a useful property if this module is provided by a third party in compiled form and the source code is not available.

Capability safety also allows developers to reason about global invariants about the use of resources, while only needing to inspect part of the program. For example, to verify that the entire program only accesses the file system to write to log files, we first inspect the top-level code and observe that the `fileIO` resource is only passed to the `wordProcessor` module. We then inspect `wordProcessor` and observe that it passes the `fileIO` module exclusively into the `logger` module. Examining the `logger`'s code, we see that it enforces the desired invariant of writing only to log files and does not provide clients with any means of accessing `fileIO`'s functionality. Since capabilities are *non-transitive* and neither `wordProcessor` nor `logger` expose `fileIO` via their methods, it is guaranteed that, besides `wordProcessor` and `logger`, no other program module can access the `fileIO` module. It is unnecessary to inspect any other modules, which could make up an arbitrarily large fraction of the program, because we can rely on the capability-

$\boxed{cap(l,e,\mu)}$ $\boxed{cap_{store}(l,\mu)}$ $\boxed{cap_{stack}(l,e,\mu)}$

$$\frac{}{cap(l,e,\mu) = cap_{store}(l,\mu) \cup cap_{stack}(l,e,\mu)} \text{ (CAP-CONFIG)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu}{cap_{store}(l,\mu) = pointsto(l,\mu) \cup pointsto(\overline{d},\mu)} \text{ (CAP-STORE)}$$

$$\frac{l.m(l') \rhd e' \notin e}{cap_{stack}(l,e,\mu) = \varnothing} \text{ (CAP-STACK-NOCALL)}$$

$$\frac{l.m'(l'') \rhd E' \notin E}{cap_{stack}(l, E[l.m(l') \rhd e'], \mu) = pointsto(e',\mu) \cup cap_{stack}(l,e',\mu)} \text{ (CAP-STACK)}$$

Figure 2.12: cap rules.

safety property to ensure that those parts of the program can never acquire a capability to access `fileIO`.

Thus, our approach enables reasoning that is impossible in conventional languages, such as Java, without a global analysis that requires access to all code in the program, or use of the Java security manager (which is difficult to use correctly due to its excessive complexity [13, 39]).

2.6.2 Formal Definition of Capability Safety

To formalize capability safety, we must first present a formal notion of capabilities. Our definition of capabilities is given by two sets of rules—the cap and $pointsto$ rules. Intuitively, $pointsto$ captures references between objects, while cap is a higher-level relation that builds on $pointsto$ to define capabilities. We describe the rules, give an example of how the rules are applied, state the capability-safety theorem, and finally prove Wyvern capability safe.

cap Rules

The capabilities of an object are determined according to the functions and rules in Figure 2.12. Intuitively, our definition of capabilities has two parts. The first part, cap_{store}, captures the principals that an object has a reference to in the heap, either as one of its fields, or as a location captured in one of its methods (which act as closures in Wyvern). The second part, cap_{stack}, is more subtle: it captures the principals that an object has a reference to in an on-the-fly execution of one of the object's methods. More formally:

- $cap(l,e,\mu)$ takes a location l, an expression e, and a store μ, and returns a set of locations identifying principals that constitute the total set of capabilities of an object identified by l when a program e is being executed in the context of memory μ.
- $cap_{store}(l,\mu)$ takes a location l and a store μ and returns a set of locations identifying principals to which an object identified by l has direct access by virtue of the object's static state in the store μ. In other words, the function determines the object's capabilities that can be statically deduced by examining the code stored in the object.

- $cap_{stack}(l, e, \mu)$ takes a location l, an expression e, and a store μ, and returns a set of locations identifying principals to which an object identified by l has direct access by virtue of the execution state of methods of l executing in e in the context of memory μ. That is, the function determines the object's capabilities acquired on the stack.

Since, in the process of evaluation, methods may have received new principals as arguments and method bodies may have been re-written to include new principals, the sets returned by $cap_{store}(l, \mu)$ and $cap_{stack}(l, e, \mu)$ may differ.

The CAP-CONFIG rule defines the relation between the three functions: the total set of capabilities of an object consists of capabilities it has statically from the code it stores and capabilities it acquired on execution. The CAP-STORE rule defines $cap_{store}(l, \mu)$. It requires the object identified by l to be in the store μ and returns two sets of locations identifying principals to which an object identified by l has direct access via itself and its declarations.

The CAP-STACK-NOCALL and CAP-STACK rules define $cap_{stack}(l, e, \mu)$. The CAP-STACK-NOCALL rule is used when there are no method-call stack frames with the receiver l on the stack ($l.m(l') \rhd e' \notin e$) and returns an empty set, as in such cases, l aquires no capabilities from executing e. If the stack contains method-call stack frames where the receiver is l, the CAP-STACK rule is used, and the capabilities are "collected" from the outermost such method-call stack frame (i.e., the furthest method-call stack frame from the expression that is being evaluated) up to the expression being evaluated. The condition $l.m'(l'') \rhd E' \notin E$ means that there must be no method-call stack frames with l as the receiver preceding the method call in consideration, which assures that, as we go down the stack, we do not miss any method calls with l as a receiver. The $cap_{stack}(l, e, \mu)$ returns a set of locations identifying the principals that the method body contains and the principals that l can access on the rest of the stack.

pointsto Rules

Capabilities functions use *pointsto* functions (Figure 2.13). The *pointsto* functions take an expression e, a declaration d, or a list of declarations $\overline{d}$ and a store μ, and return a set of locations identifying principals to which the expression, the declaration, or the list of declarations point (i.e., have direct access) in the context of memory μ.

A variable does not point to any location (POINTSTO-VAR). A new expression points to locations to which the new object's declarations points (POINTSTO-NEW). A method, an object field and its assignment, as well as a bind construct (POINTSTO-METHOD, POINTSTO-FIELD, POINTSTO-ASSIGN, and POINTSTO-BIND respectively) point to locations in their subexpressions. Depending on whether a location is identifying a principal or a pure object, it points to either itself (POINTSTO-PRINCIPAL) or nothing (POINTSTO-PURE) respectively. Depending on whether the method caller is a principal or a pure object, a method-call stack frame points to either itself (POINTSTO-CALL-PRINCIPAL) or a set of locations pointed to by the method body (POINTSTO-CALL-PURE) respectively.

POINTSTO-PRINCIPAL and POINTSTO-PURE look similar to $cap_{store}(l, \mu)$, but differ semantically: in these *pointsto* rules, l is treated as an expression, not as a location identifying a principal, and so the only location l can access is itself.

A list of declarations points to a union of sets of locations to which each declaration in the list points (POINTSTO-DECLS). A method declaration points to the locations to which the

$\boxed{pointsto(e,\mu)}$ $\boxed{pointsto(\overline{d},\mu)}$ $\boxed{pointsto(d,\mu)}$

$$\frac{}{pointsto(x,\mu) = \varnothing}\ \text{(POINTSTO-VAR)}$$

$$\frac{}{pointsto(\texttt{new}_s(x \Rightarrow \overline{d}),\mu) = pointsto(\overline{d},\mu)}\ \text{(POINTSTO-NEW)}$$

$$\frac{}{pointsto(e.m(e'),\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)}\ \text{(POINTSTO-METHOD)}$$

$$\frac{}{pointsto(e.f,\mu) = pointsto(e,\mu)}\ \text{(POINTSTO-FIELD)}$$

$$\frac{}{pointsto(e.f = e',\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)}\ \text{(POINTSTO-ASSIGN)}$$

$$\frac{}{pointsto(\texttt{bind}\ x = e\ \texttt{in}\ e',\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)}\ \text{(POINTSTO-BIND)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu}{pointsto(l,\mu) = \{l\}}\ \text{(POINTSTO-PRINCIPAL)} \qquad \frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{pure}} \in \mu}{pointsto(l,\mu) = \varnothing}\ \text{(POINTSTO-PURE)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu}{pointsto(l.m(l') \rhd e,\mu) = \{l\}}\ \text{(POINTSTO-CALL-PRINCIPAL)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{pure}} \in \mu}{pointsto(l.m(l') \rhd e,\mu) = pointsto(e,\mu)}\ \text{(POINTSTO-CALL-PURE)}$$

$$\frac{}{pointsto(\overline{d},\mu) = \cup \bigcup_{d \in \overline{d}} pointsto(d,\mu)}\ \text{(POINTSTO-DECLS)}$$

$$\frac{}{pointsto(\texttt{def}\ m(x:\tau_1):\tau_2 = e,\mu) = pointsto(e,\mu)}\ \text{(POINTSTO-DEF)}$$

$$\frac{}{pointsto(\texttt{var}\ f:\tau = x,\mu) = \varnothing}\ \text{(POINTSTO-VARX)}$$

$$\frac{}{pointsto(\texttt{var}\ f:\tau = l,\mu) = pointsto(l,\mu)}\ \text{(POINTSTO-VARL)}$$

Figure 2.13: *pointsto* rules.

method body points (POINTSTO-DEF). A field declaration points to locations to which the field's value points: if the field's value is a variable, the field declaration does not point to any location (POINTSTO-VARX), and if the field's value is a location, the field declaration points to the same location as the value location (POINTSTO-VARL).

In our system, capabilities are non-transitive for principal objects and transitive for pure objects to which a principal points. As pure objects do not have fields, they cannot point to any resources and their methods cannot capture resources. Thus, POINTSTO-PRINCIPAL and POINTSTO-PURE do not involve declarations of the object identified by the location (cf. POINTSTO-NEW). However, an executing method of a pure object can have resources in it if they

were passed as arguments. Since the pure object cannot own the resource arguments, in this case, the capabilities are transitive, and the resource arguments are owned by the resource caller down the stack. Therefore, POINTSTO-CALL-PRINCIPAL considers only the principal caller, whereas POINTSTO-CALL-PURE allows a principal caller down the stack to have capabilities to access principals in a pure callee's method.

Determining Capabilities of an Object

To demonstrate how capabilities of an object are determined, consider the following definition of the `prettyChart` module:

```
module def prettyChart(logger: Logger): WordCloud
  def updateLog(entry: String): Unit
    logger.appendToLog(entry)
```

Assume that the definition of the `logger` module is as in Figure 2.2 and that the last line in the above code snippet is currently being executed, i.e., the method `appendToLog` is called on the `logger` object. The `logger` object in the store μ looks like:

$$l_{logger} \mapsto \{\ x \Rightarrow \texttt{def}\ \ appendToLog(entry : String) : Unit \\ l_{io}.getStandardLogFile().append(entry)\ \}_{\texttt{resource}}$$

To find the capabilities l_{logger} has statically, i.e., from the code it contains, we apply CAP-STORE, POINTSTO-PRINCIPAL, POINTSTO-DEF, POINTSTO-METHOD, POINTSTO-PRINCIPAL, and POINTSTO-VAR as follows:

$$\begin{aligned} cap_{store}(l_{logger}, \mu) &= pointsto(l_{logger}, \mu) \cup pointsto(\texttt{def}\ appendToLog(...)\ ..., \mu) \\ &= \{l_{logger}\} \cup pointsto(\texttt{def}\ appendToLog(entry : String) : Unit \\ &\qquad\qquad l_{io}.getStandardLogFile().append(entry), \mu) \\ &= \{l_{logger}\} \cup pointsto(l_{io}.getStandardLogFile().append(entry), \mu) \\ &= \{l_{logger},\ l_{io}\} \end{aligned}$$

To find the capabilities l_{logger} acquired on the stack, we use CAP-STACK, CAP-STACK-NOCALL, POINTSTO-METHOD, POINTSTO-PRINCIPAL, and POINTSTO-VAR as follows:

$$\begin{aligned} &cap_{stack}(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \rhd l_{io}.getStandardLogFile().append(entry)], \mu) \\ &\qquad = pointsto(l_{io}.getStandardLogFile().append(entry), \mu) \\ &\qquad \cup cap_{stack}(l_{logger}, l_{io}.getStandardLogFile().append(entry), \mu) \\ &\qquad = pointsto(l_{io}.getStandardLogFile().append(entry), \mu) \\ &\qquad = \{l_{io}\} \end{aligned}$$

Finally, by CAP-CONFIG, the total set of capabilities of l_{logger} when executing the `appendToLog` method is

$$\begin{aligned} &cap(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \rhd l_{io}.getStandardLogFile().append(entry)], \mu) \\ &\quad = cap_{store}(l_{logger}, \mu) \\ &\quad \cup cap_{stack}(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \rhd l_{io}.getStandardLogFile().append(entry)], \mu) \\ &\quad = \{l_{logger},\ l_{io}\} \end{aligned}$$

As expected, l_{logger} has capabilities to access l_{io} and no other resource object.

This way, the *cap* and *pointsto* rules allow us to determine capabilities of every object on every step of execution, which serves as a basis for our formal system and the capability-safety proof.

Capability-Safety Theorem

We now state the capability-safety theorem (formerly authority-safety theorem [43, 44]) formally.

Theorem (Capability Safety)**.** *If*

1. $\Gamma \mid \Sigma \vdash^{e'''} e : \tau$,
2. $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$,
3. $l_0 \mapsto \{x \Rightarrow \overline{d_0}\}_{\texttt{resource}} \in \mu'$,
4. $l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu$, *and*
5. $cap(l, e', \mu') \setminus cap(l, e, \mu) \supseteq \{l_0\}$,

then one of the following must be true:

1. ***Object creation:***
   (a) $e = E[l.m(l') \rhd E'[\texttt{new}_{\texttt{resource}}(x \Rightarrow \overline{d_0})]]$ *and*
   (b) $e' = E[l.m(l') \rhd E'[l_0]]$, *where*
   (c) $\forall l_a.m_a(l'_a) \rhd E'' \in E'$, $l_a \mapsto \{x \Rightarrow \overline{d_a}\}_{\texttt{pure}} \in \mu$
2. ***Method call:***
   (a) $e = E[l.m(l_0)]$,
   (b) $e' = E[l.m(l_0) \rhd [l_0/y][l/x]e'']$, *and*
   (c) $y \in e''$
3. ***Method return:***
   (a) $e = E[l.m(l') \rhd E'[l_a.m_a(l'_a) \rhd l_0]]$ *and*
   (b) $e' = E[l.m(l') \rhd E'[l_0]]$, *where*
   (c) $\forall l_b.m_b(l'_b) \rhd E'' \in E'$, $l_b \mapsto \{x \Rightarrow \overline{d_b}\}_{\texttt{pure}} \in \mu$

The formal statement of capability safety makes the informal statement above more precise, in that:

1. The principal acquiring capabilities in the given evaluation step must be a receiver of a method-call stack frame on the stack, but not necessarily the immediate receiver for the expression under evaluation.
2. Receivers of all method-call stack frames between the principal receiver and the expression under evaluation must be pure.

These points allow us to define capability safety comprehensively, while treating pure objects, essentially, as a part of the principal that uses them. Below is a sketch of the proof of the capability-safety theorem, while the full proof is presented in Appendix A.2.4.

Proof Sketch. The proof is by induction on a derivation of $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. Essentially, we need to determine the difference in principal l's capabilities between the two states. We start by considering E-CONGRUENCE. Through several lemmas, presented in Appendix A.2.3, we

found that l's position on the call stack is important for how we determine the difference in its capabilities and that there are two cases (this fact is formally stated and proven in Lemma 8):

- If there are only pure callers after the last method-call stack frame where l is a caller, i.e., l was the last principal caller on the stack, then

$$cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu) = cap_{store}(l, \mu') \cup pointsto(e', \mu') \cup cap_{stack}(l, e', \mu') \setminus cap_{store}(l, \mu) \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu)$$

- Otherwise, if the last method-call stack frame where l is the caller is followed by a method-call stack frame with a principal caller that is not l, or if the stack has no method-call stack frames with principal callers, then

$$cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu) = cap_{store}(l, \mu') \cup cap_{stack}(l, e', \mu') \setminus cap_{store}(l, \mu) \cup cap_{stack}(l, e, \mu)$$

In either case, the change in l's capabilities depends on expressions e and e' inside the evaluation context E and thus on $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. So next, we consider all possible terminal-form reduction steps and, using the *cap* and *pointsto* rules, calculate the difference in capabilities of the principal before and after the reduction step.

In the cases for E-NEW, E-METHOD, and E-STACKFRAME, l's capabilities increase, which matches the three situations stated in the theorem. The rest of the reduction rules do not cause any capability gains. □

2.7 Implementation

We implemented the module system as part of the open-source Wyvern compiler and interpreter, which is available on GitHub: `https://github.com/wyvernlang/wyvern`. Examples from Figures 2.2 and 2.3 run as part of the `wyvern.tools.tests.Figures` test suite and can be found in the `tools/src/wyvern/tools/tests/figs` subdirectory of the project.

2.8 Limitations

Our threat model makes an important assumption that the code in the trusted code base of a software system is trustworthy. We assume that the security analysts who are in charge of the trusted code base are honest and do not make mistakes. This may not be true in practice, and thus our approach is susceptible to insider attacks, which are common to systems that reason about trusted code bases and involve vulnerabilities inside the trusted code base.

For example, a security analyst responsible for the trusted code base may have a malicious intent and subvert the software system by exporting the functionality of system resources via wrapper functions. A wrapper function is a function of a module (e.g., `logger`) that "wraps" the functionality of a function of another module (e.g., a module of type `FileIO`), performing the same operations as the original function, e.g.:

```
module def logger(io: FileIO): Logger
  def write(fileName: String, text: String)
    io.write(fileName, text)
```

By calling `logger`'s `write` method, an extension importing `logger` could write to any file in the file system, and this would not be exposed in the `logger`'s type or interface. In a similar fashion, the malicious `logger` module may export functionality of an entire file I/O module, potentially changing function names to obfuscate the exposure. In such a case, an extension that is allowed to import `logger` would, in essence, have capabilities to access a module of type `FileIO`.

Although insider attacks directed at the trusted parts of a system are beyond our reach, our approach allows software developers to formally reason about the isolation of security- and privacy-related resources in a software system and gives software developers a tool to enforce certain isolation properties. Also, the described limitations can be mitigated either by using more rigorous software development practices, e.g., code reviews, for critical parts of the system, or by complementing our approach with more complex analyses, e.g., by using an effects system (discussed in Chapter 3) or an information flow analysis.

2.9 Related Work

The object-capability model, in which capabilities guard access to objects, was introduced by Miller [48]. The two pioneering programming languages that use object capabilities are E [47] and W7 [62]. Our approach carries forward this line of work by exploring a statically typed, capability-safe programming language and providing support for modules as capabilities.

Similar to all object-capability languages, Wyvern is inherently more resistant to confused-deputy attacks [61]. In Wyvern, all uses of capabilities are explicit in code, thus enabling the deputy to require that the requester provide the necessary capability before the deputy can perform the requested operation. Furthermore, Wyvern improves on other capability-based systems such as E [47] because capabilities in Wyvern are statically typed, making it more obvious from looking at the objects' types which objects are security- and privacy-related (i.e., which objects are capabilities) and which objects are "harmless" and can be ignored.

Our module-system design was primarily inspired by the capability-passing modules design in Newspeak [8] and its predecessors, such as MzScheme's Units [25]. As in Newspeak, Wyvern modules are first-class. However, our static types support reasoning about capabilities based on module interfaces (Newspeak is dynamically typed), and our approach reduces the overhead of ubiquitous module parameterization by allowing pure modules to be directly imported, rather than passed in as arguments (in Newspeak, all module dependencies must be passed in as arguments).

Several research efforts limited mainstream, non-object-capability programming languages to turn them into object-capability languages. Typically the imposed restrictions disallow mutable global state (e.g., static fields), limit access to certain APIs (e.g., the reflection API), and prohibit ambient authority [73]. Sometimes sandboxing is used to facilitate isolation of program components (e.g., add-ons). Programming languages in this category include: Joe-E [45] (a restricted subset of Java in which program privileges are represented using object references), work by Hayes et al. [27] (a restricted version of Java in which capabilities are represented as a special type of interface), Emily [68] (a performant subset of OCaml), CaPerl [31] (a subset of modified Perl), Oz-E [67] (a proposed variation of Oz), and Google's Caja [26, 49] (an object-capability-based subset of JavaScript). In contrast, our work explores a module system with explicit support

for object-capabilities without the constraint of adapting an existing language, enabling a cleaner design.

SHILL [50] is a secure shell scripting programming language that takes a declarative approach to access control. In SHILL, capabilities are used to control access to system resources, contracts are used to specify what capabilities each script requires, and capability-based sandboxes are used to enforce contracts at run time. SHILL supports compositional reasoning by tracing capabilities through program invocations and, if necessary, attenuating capabilities on every transition. The authority of the SHILL program's entry point is ambient, but its transition to other parts of the program is limited via contracts and sandboxes. SHILL does not include mutable state (e.g., variables), which are part of our model and make our notion of capability safety more interesting; nor does SHILL include a module system.

Monte [1] is a capability-safe programming language that aims to support secure distributed computing. Similar to Wyvern, Monte's modules are first-class values and are translated into objects. In contrast to Wyvern, Monte is dynamically typed, which means that Monte's compiler offers no help in reasoning about capabilities. In addition, Monte does not differentiate between security- and privacy-crucial modules (Wyvern's resource modules) and other modules (Wyvern's pure modules), and all imported modules are immutable, which prevents programmers from using module-local state and thus limits the language expressivity.

Maffeis et al. [41] formalized the notions of capability and authority safety and proved that capability safety implies authority safety, which in turn implies resource isolation. They showed that these semantic guarantees hold in a Caja-based subset of JavaScript and other object-capability languages. Maffeis et al.'s formal system defines capabilities topologically: objects are represented as nodes in a graph, and a path between two nodes implies that the source node can access the destination node. Such a definition renders capabilities transitive. In contrast, our formal definition of capabilities is non-transitive, enabling reasoning about the attenuation of capabilities.

Devriese et al. [16] presented an alternative formalization of capability safety that is based on logical relations. They argue that formalizations like Maffeis et al.'s [41] are too syntactic and the topological definition of authority is insufficient to characterize capability safety as it leads to over-approximation of authority. Our non-transitive definition of capabilities is similarly more precise than prior, transitive topological definitions. However, our focus is on a relatively simple (compared to logical relations) type system that provides capability safety with respect to this more refined notion of capabilities, along with support for modules as capabilities.

Another line of related work assumes a capability-safe base language and develops logics or advanced type systems to state and prove properties that are built on capabilities. Drossopoulou et al. analyzed Miller's mint and purse example [48], rewrote it in Joe-E [19] and Grace [53], and based on their experience, proposed and refined a specification language to define policies required in the mint and purse example [20, 21, 22, 23]. Also, Dimoulas et al. [17] proposed a way to extend an underlying capability-safe language with declarative access control and integrity policies for capabilities, and proved that their system can soundly enforce the declarative policies. Dimoulas et al.'s formalization, like that of Maffeis et al. but unlike ours, formalizes access capabilities transitively.

Chapter 3

Authority Safety via Effects

An effect system can be used to reason about various aspects of a program execution, including exceptions (e.g., Eiffel's exceptions and the widely used Java's checked exceptions [29]), memory effects [38], concurrency [7, 11, 18], and security [71]. Our effect system, which is implemented as part of Wyvern, tracks the use of system resources, such as the file system, network, and keyboard, and is intended to help developers reason about how application modules use these resources, i.e., modules' authority. Wyvern's effect system is built on top of and relies on features of Wyvern's module system (Chapter 2).

3.1 Running Example

Drawing inspiration from a recent report on security vulnerabilities in text editors [4],[1] we use a text-editor application as a running example to demonstrate the key features of our effect-system design. The overall architecture of this application is shown in Figure 3.1. Each box in the diagram represents a module, and the arrows represent module imports. For the purposes of our forthcoming examples, the solid arrows are imports that take place, and the dashed arrows represent potential imports that may or may not occur.

The application is written using Wyvern's libraries, which contain modules representing system resources, such as the file system and network. These modules rely on access to native backend modules, such as `java` and `python`, which are Wyvern's Java and Python backends, respectively. In the text editor, we focus only on the `logger` module that implements the logging facility of the application. The text editor allows supplementing its core functionality with various third-party plugins. We assume that the application requires that all plugins and user-facing modules of the text editor itself update the log file with the user-observable actions they perform. In our examples, we use two sample plugins: one that, as the user types in code, detects code patterns and offers to complete the code for them, and another that analyzes the text editor's log file and provides insight into how the text-editor application is used.

[1]Notably, it is possible to prevent the vulnerabilities described in the report by implementing a text editor using an object-capability-based programming language similar to Wyvern, which facilitates the enforcement of the principle of least privilege.

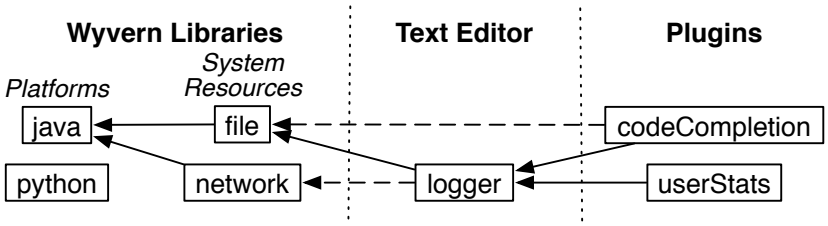


Figure 3.1: The overall architecture of the text-editor application. Boxes represent modules, and the arrows represent module imports. The solid arrows are imports that take place, and the dashed arrows represent potential imports that may or may not occur.

```
1  resource type Logger
2    effect ReadLog
3    effect UpdateLog
4    def readLog(): {this.ReadLog} String
5    def updateLog(newEntry: String): {this.UpdateLog} Unit
6
7  module def logger(f: File): Logger
8  effect ReadLog = {f.Read}
9  effect UpdateLog = {f.Append}
10 def readLog(): {ReadLog} String
11   f.read()
12 def updateLog(newEntry: String): {UpdateLog} Unit
13   f.append(newEntry)
```

Figure 3.2: A type and a module implementing the logging facility in the text-editor application.

The dashed vertical lines represent the conceptual boundaries between parts of the application that vary in the level of trust based on the security of the contained code. Modules in the Wyvern libraries are the most trusted since they provide functionality essential for all applications developed in Wyvern and were written with security in mind. Modules of the text-editor application are less trusted since they are more likely to contain fallible code. Finally, the plugins are the least trusted since they are written by third parties and may be error-prone, vulnerable to exploitation, or outright malicious.

3.2 Wyvern Effects Basics

Consider the code in Figure 3.2 that shows a type and a module implementing the logging facility of the text-editor application. The keyword `resource` in the type definition indicates that the implementations of this type may have state and may access system resources. In the given implementation of the `Logger` type, the `logger` module accesses the log file, which is a resource. 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: the `ReadLog` effect that, when run, the `readLog`

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

Figure 3.3: The type of the file resource.

method produces and the `UpdateLog` effect that, when run, the `updateLog` method produces. These effects are abstract because they do not have a definition, and it is up to the module implementing the `Logger` type to define what they mean. The effect names are flexible, and the software developer may choose effect names that are most meaningful for a given module.

The `logger` module implements the `Logger` type. Access to the file system is granted via the capability to the file object, which is of type `File` (shown in Figure 3.3) and is passed into `logger` as a parameter. The `logger` module's effects 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` module, and accordingly, the `readLog` method, which produces the `ReadLog` effect, calls `file`'s `read` method. Similarly, the `UpdateLog` effect of the `logger` module is defined to be the `Append` effect of the `file` module, and accordingly, the `updateLog` method, which produces the `UpdateLog` effect, calls `file`'s `append` method. In general, effects in a module must always be concrete, and effects in a type may be either abstract or concrete.

Effects are members of objects (with modules as an important special case), 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 `file` 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 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, 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`.

3.2.1 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 our language's 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 3.3.2).

3.3 Software Development Patterns Facilitated by Wyvern's Effect System

In this section, we present a selected set of patterns that could be used by software architects and security analysts to ensure the security of the software system under development.

3.3.1 Controlling Operations Performed on Resources

Our design of effects in Wyvern allows software developers, software architects, and security analysts to control what operations are performed on system resources and other resource-containing modules in a software system written in Wyvern.

Consider the two previously introduced plugins for the text editor. As we pointed out earlier, these plugins lie outside the trusted code base for the application because they were written by third parties and may contain bugs, which could introduce vulnerabilities, or be actively malicious. Thus, to better maintain security of the text-editor application and minimize any potential damage from the plugins, developers of the text editor need to control what resources the plugins may access and what operations they are allowed to perform on those resources. The first part of this task, i.e., controlling access to resources, is done via Wyvern's capability-based module system (Chapter 2), which limits the plugins' access to resource modules. Briefly, the plugins may use capabilities that are passed into them on creation, i.e., as parameters to their module functors, or that become available via method calls. The second part of the task is limiting what

```
module def codeCompletion(log: Logger)
def findTemplate(wordSequence: String): {log.UpdateLog} String
  ...
  log.updateLog("Searching for a matching template.")
  ...
  log.updateLog("Found matching template.")
  ...

module def userStats(log: Logger)
def calculateUserStats(): {log.ReadLog, log.UpdateLog} String
  ...
  log.updateLog("Starting to analyze the log content.")
  analyzeLogContent(log.readLog())
  ...
```

Figure 3.4: Excerpts from the code-completion and user-statistics-analyzer plugins of the text-editor application.

operations are performed on the capabilities the plugins have access to, and this is the main focus of Wyvern's effect system.

Relevant code excerpts from the two plugins are shown in Figure 3.4. Both plugins have access to the `logger` module, which is passed in as a functor parameter; however, they use it differently. Both plugins must follow the text editor's policy of recording user-observable actions they perform, but only the `userStats` plugin needs to perform more operations on `logger` than simply updating it. The `codeCompletion` module needs `logger` only to update the log file about the status of the search of an appropriate template in its `findTemplate` method. On the other hand, along with simply updating the log file, the `userStats` module reads the log file to analyze its content. Accordingly, `codeCompletion`'s `findTemplate` method must only call `logger`'s `updateLog` method and must have only the `log.UpdateLog` effect, whereas `userStats`'s `calculateUserStats` method may call both `logger`'s `updateLog` and `readLog` methods and may have both the `log.ReadLog` and `log.UpdateLog` effects.

Wyvern's effect system ensures that the method bodies of `findTemplate` and `calculateUserStats` methods produce only the effects with which the methods are annotated (more details on this are in Section 3.4). Then, a software architect or a security analyst can rely on the modules' interfaces and, specifically, the methods' effect annotations to reason about the effects that methods may produce on resources. For example, if the `codeCompletion` module's `findTemplate` method calls `log.readLog()`—erroneously or on purpose—Wyvern's compiler will report an error saying that the method's effect annotations do not reflect the effects produced in the method body. Consequently, it is sufficient for a security analyst to examine only `codeCompletion`'s interface, but not its code, to verify that its code performs only the allowed operations on the logging resource. This allows a software architect or a security analyst to control what operations are performed on the important resource modules of an application and also significantly simplifies the reasoning process when a security analyst or a software architect performs an analysis of the application security, as the method effect annotations reliably reflect the operations performed on resources inside the method's body.

```
1 module def remoteLogger(net: Network): Logger
2 effect ReadLog = {net.Receive}
3 effect UpdateLog = {net.Send}
4 def readLog(): {ReadLog} String
5   net.receive()
6 def updateLog(newEntry: String): {UpdateLog} Unit
7   net.send(newEntry)
```

Figure 3.5: An alternative implementation of the `Logger` type from Figure 3.2.

Another feature of Wyvern's effect-system design is that it is possible to reduce the effect-annotation overhead by aggregating several effects into one. The way the `userStats` module is written in Figure 3.4 is somewhat too verbose in that, if some other module calls `userStats`'s `calculateUserStats` method, it has to annotate the calling method with two effects. Because more code may add more effects, larger software systems might experience a snowballing of effects, when method annotations have numerous effects in them. Alternatively, the `userStats` module can be written as:

```
module def userStats(log: Logger)
effect AnalyzeLog = {log.ReadLog, log.UpdateLog}
def calculateUserStats(): {AnalyzeLog} String
  ...
```

In this version, the `userStats` module declares the `AnalyzeLog` effect which, in its definition, aggregates the two logger effects. Using this version of `userStats`, any method that calls `calculateUserStats` would have to add only one extra effect annotation, instead of two, thus reducing the effect-annotation overhead.

3.3.2 Information Hiding and Polymorphism

Having been introduced by Parnas in the early 1970s [57, 58], the principle of information hiding is a key software development principle that states that, in a software application, implementation details of a particular software module should be hidden behind a stable interface. This principle promotes modularity in the software implementation and gives software developers more flexibility to modify the existing implementation of a module without affecting other modules. Our design facilitates the principle of information hiding.

Figure 3.5 shows an alternative implementation of the `Logger` type from Figure 3.2. In this version, the log file is stored on some remote machine, and the network resource (instead of the file system resource) is used to perform operations on the log. Importantly, the `Logger` type contains no information about what resource should be used to implement the logging functionality, and thus, a module implementing the `Logger` type may use any resource or no resources at all (in which case `Logger`'s effects could be defined as empty effects, i.e., `{}`). Yet the client modules that use a resource of type `Logger`, such as the two plugins discussed in the previous subsection, observe no difference in the logging functionality. The software architect may swap one `logger` version for the other or modify the implementation at any time without affecting the modules using `logger`, provided that the interface of the `Logger` type remains the same. Thus,

using effect abstraction in the `Logger` type facilitates the principle of information hiding.

Effect members also naturally support effect polymorphism, following an idiom that has been used in Scala and other languages with type members. We designed an intuitive syntax for effect-polymorphic functions. For example, the following higher-order function can be used to invoke a function with an arbitrary effect:

```
def invokeTwice[effect E](f: Unit -> {E} Unit)
  f()
  f()

invokeTwice[log.UpdateLog]( () -> log.updateLog("Updating log.") )
```

Here `invokeTwice` is parameterized by an effect `E`. The `invokeTwice` function takes another function that has no arguments and produces no result but has effect `E`, and invokes that function twice. We call `invokeTwice`, instantiate the effect parameter with `log.UpdateLog`, and give `invokeTwice` a function that updates the log file.

The compiler rewrites `invokeTwice` using only effect members. In this rewriting, the `invokeTwice` function takes an extra parameter, an `EffectHolder` object, which holds the effect parameter `E` as an effect member. The desugared code would look like this:

```
type EffectHolder
  effect E

def invokeTwice(eh: EffectHolder, f: Unit -> {eh.E} Unit)
  f()
  f()

let effectHolder: EffectHolder = new
  effect E = log.UpdateLog
in invokeTwice(effectHolder, () -> log.updateLog("Updating log.") )
```

Note that this code creates an `effectHolder` object that instantiates effect `E` with `log.UpdateLog`. We also rely on path-dependent types [3]: the second parameter of `invokeTwice` can refer to the first parameter in order to describe the effect of the argument function `f`.

3.3.3 Designating Important Resources Using Globally Available Effects

To enforce certain architecture or security constraints, it may be helpful to "highlight" effects of a particular module and "suppress" the effects of another, thus controlling what effects are propagated throughout the application code and designating effects produced on one module as more important than effects produced on another one. Using Wyvern's effect system, a software architect or a security analyst can achieve this by making effects of a module globally available.

An effect can be made globally available by defining it in a module that can be imported from anywhere in the program. Wyvern's pure modules, which are purely functional modules that do not contain any state, have this "importable-anywhere" property. (Recall that modules with state, i.e., resource modules, must be instantiated and passed into all modules that use them.) Therefore, to make effects globally available, we specify them in a pure module. For example, Figure 3.6 shows a pure module that defines effects for the `File` type.

```
module fileEffects
effect Read = {system.FFI}
effect Write = {system.FFI}
effect Append = {system.FFI}
effect Delete = {system.FFI}
...
```

Figure 3.6: A pure module defining file effects.

```
import fileEffects
resource type File
  def read(): {fileEffects.Read} String
  def write(s: String): {fileEffects.Write} Unit
  def append(s: String): {fileEffects.Append} Unit
  def delete(): {fileEffects.Delete} Unit
  ...
```

Figure 3.7: A version of the `File` type that uses globally available effects.

For convenience, we chose the effect names in the `fileEffects` module to match the effect names in the original `File` type, shown in Figure 3.3. All effects in the `fileEffects` module are defined to be the `system.FFI` effect, which is the lowest-level Wyvern effect.[2] It represents the effects on the native Wyvern backends, such as the `java` and `python` modules in Figure 3.1, and thus is the effect produced by all methods called on the `java` and `python` modules. The `system.FFI` effect is built-in and is globally available without being imported, and so, any module can use this effect to annotate its methods. (Yet to be able to actually call a method on the `java` or `python` modules, a module needs to possess an appropriate capability.) However, given that `system.FFI` indicates some effect on a Wyvern's native backend, during a security code review, observing this effect in a method annotation of a third-party plugin would prompt suspicion and necessitate further investigation.

Having made file effects globally available, the `File` type does not need to redefine its effects (although it can), and we can use the `fileEffects`' effects to annotate methods in the `File` type directly (Figure 3.7). With this change, we can also write the `Logger` type and the `logger` module differently. Figure 3.8 presents an alternative version of them. In this version, the `Logger` type imports the `fileEffects` module and uses effects defined in it in method effect annotations, instead of declaring any of its own effects. By writing the `Logger` type this way, the software architect designates that it is more important to keep track of effects produced on the file system

[2]There is a potential issue with giving all the effects the same definition because, then, one effect could be substituted for another and the effect system would be unable to differentiate among them. For example, the way `fileEffects`'s effects are defined here, if a method is annotated with the `Read` effect but in fact performs a `write` operation, the effect system would be unable to detect this issue. A solution to this problem is to make the definitions abstract either by introducing a subeffecting relationship that allows providing a definition without using precise effects or by imposing per-directory visibility restrictions so that, outside certain packages, the effect definitions are invisible and the effects are treated as abstract. The former variant of the solution is currently under development.

```
import fileEffects
resource type Logger
  def readLog(): {fileEffects.Read} String
  def updateLog(newEntry: String): {fileEffects.Append} Unit

module def logger(f: File): Logger
import fileEffects
def readLog(): {fileEffects.Read} String
  f.read()
def updateLog(newEntry: String): {fileEffects.Append} Unit
  f.append(newEntry)
```

Figure 3.8: A version of the `Logger` type and implementation that uses globally available file effects.

```
module def codeCompletion(log: Logger)
import fileEffects
def findTemplate(wordSequence: String): {fileEffects.Append} String
  ... // same as in Figure 4
```

Figure 3.9: A version of the code completion plugin that uses the alternative version of the `Logger` type from Figure 3.8.

than on the log. Then, following the `Logger` type, in the new version, the `logger` module also imports the `fileEffects` module and uses the file effects defined in it in the `logger`'s method effect annotations.

Finally, `logger`'s client modules, such as the two plugins described above, are also written differently. For example, Figure 3.9 shows a new version of the code completion plugin. Similarly to the `logger` module, `codeCompletion` now imports the `fileEffects` module and uses its effects to annotate `codeCompletion`'s methods, thus, "skipping" a dependency level and exposing the information that `codeCompletion`'s `findTemplate` method produces an effect on the file system.

Another notable outcome of using globally available effects like this is that, if `codeCompletion` already uses `file`, e.g., to store custom, user-defined templates in a file, written this way, it can use `fileEffects`'s effects to annotate methods that only use `logger`, methods that only use `file`, and methods that use both without having to introduce any new effects.

Thus, a software architect or a security analyst can designate effects on the resources that are more important to track than others by making the effects of the more important resource globally available and not declaring any effects in the type describing the important resource. For example, a software architect can establish that it is more important to track effects on the `file` resource than on `logger` by creating the pure `fileEffects` module to represent the effects of the `File` type and not declaring any effects in the `File` type itself. Generally, making effects of any resource globally available promotes the use of those effects throughout the application code. In turn, this allows for better tracking of how resources are used, which is beneficial when reasoning about the architecture and security of an application.

3.3.4 Authority Attenuation

When performing a security analysis of an application, an important component of privilege is operations performed on a resource being accessed. In the field of software security, such operations represent *authority* over the accessed module [48].[3] For example, the `logger` module is expected to perform the read and append operations on the log file; however, it is not supposed to completely overwrite the log file. In other words, `logger` should have authority to read and append to the log file but should not have authority to overwrite it.

Notably, Wyvern effects that describe operations performed on modules are a good medium for representing authority over modules. For example, the fact that the `logger` module's effects use only `file`'s `Read` and `Append` effects in `logger`'s effect definitions signifies that the only operations `logger` performs on the log file are the read and append operations, meaning that the only authority `logger` has over the log file is to read it and append to it.

Furthermore, our effect-system design allows expressing the notion of *authority attenuation*, which is a common software-security pattern [51]. Authority attenuation happens when the original set of operations that can be performed on a resource is limited by an intermediary object [48]. For example, consider the sequence of module dependencies from Figure 3.1 consisting of the `file` module, the `logger` module, and the `codeCompletion` module. There are several operations that can be performed on a file (at least the four shown in the `File` type in Figure 3.3), but `logger` performs only two of them (as was mentioned above and as can be seen from its effects' definitions in Figure 3.2). The `codeCompletion` module can access the `logger` module but not the `file` module, and so, the only operations it can perform on `file` are those that `logger` can perform. Thus, the `logger` module attenuates `codeCompletion`'s authority over the `file` module.

Therefore, Wyvern's effect system aids software architects and security analysts in observing and establishing the authority attenuating relationship between modules of a software application, which may be desired and beneficial during the design phase of a software application, a security audit, or an architecture review of a software application.

3.4 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 (Section 2.5.3), and here we provide only some intuition behind it. In this section, we start with describing the syntax of Wyvern's object-oriented core with effects, then present an example of the module-to-object translation, followed by a description of Wyvern's static and dynamic semantics and subtyping rules. Finally, we state the progress and preservation theorems.

$$
\begin{array}{lcll}
n & ::= & x \mid l & \textit{names} \\
e & ::= & n & \textit{expressions} \\
 & \mid & \texttt{new}_s(x \Rightarrow \overline{d}) & \\
 & \mid & e.m(e) & \\
 & \mid & e.f & \\
 & \mid & e.f = e & \\
s & ::= & \texttt{resource} \mid \texttt{pure} & \\
\varepsilon & ::= & \overline{n.g} & \textit{effects} \\
d & ::= & \texttt{def}\ m(x : \tau) : \{\varepsilon\}\ \tau = e & \textit{declarations} \\
 & \mid & \texttt{var}\ f : \tau = n & \\
 & \mid & \texttt{effect}\ g = \{\varepsilon\} & \\
\tau & ::= & \{x \Rightarrow \overline{\sigma}\}_s & \textit{object type}
\end{array}
$$

$$
\begin{array}{lcll}
\sigma & ::= & \texttt{def}\ m(x : \tau) : \{\varepsilon\}\ \tau & \textit{declaration types} \\
 & \mid & \texttt{var}\ f : \tau & \\
 & \mid & \texttt{effect}\ g & \\
 & \mid & \texttt{effect}\ g = \{\varepsilon\} & \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x : \tau & \textit{var. typing context} \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s & \textit{store} \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l : \tau & \textit{store typing context} \\
E & ::= & [\,] & \textit{evaluation context} \\
 & \mid & E.m(e) & \\
 & \mid & l.m(E) & \\
 & \mid & E.f & \\
 & \mid & E.f = e & \\
 & \mid & l.f = E &
\end{array}
$$

Figure 3.10: Wyvern's object-oriented core syntax.

3.4.1 Core Syntax

Figure 3.10 shows the effect-system version of the syntax of Wyvern's object-oriented core. It is similar to the object-oriented core presented in Section 2.5.2 with a few differences that we highlight here.

In this version, 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 (since locations are a run-time notion), 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 reality, 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 (the encoding for which is the same as was shown in Section 2.5.3) that defines the variable to be used in the field initialization.

Effects in method annotations and effect-member definitions are surrounded by curly braces to visually indicate that they are sets. 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. During run time, variables in effects are substituted with locations corresponding to the values of the variables then.

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).

```
let logger = new_resource(x ⇒
  def apply(f : File) : {} Logger
      new_resource(_ ⇒
        effect ReadLog = {f.Read}
        effect UpdateLog = {f.Append}
        def readLog() : {ReadLog} String
            f.read()
        def updateLog(newEntry : String) : {UpdateLog} Unit
            f.append(newEntry)
        )
) in ...// calls logger.apply(...)
```

Figure 3.11: A simplified translation of the `logger` module from Figure 3.2 into Wyvern's object-oriented core.

3.4.2 Modules-to-Objects Translation

Figure 3.11 presents a simplified translation of the `logger` module from Figure 3.2 into Wyvern's object-oriented core (for a full description of the translation mechanism, refer to Section 2.5.3). 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.

3.4.3 Well-Formedness Rules

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 Figure 3.12. The three judgements read that, in the variable typing context Γ and the store typing context Σ, 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. Finally, an effect-definition type is well formed if the effect set in its right-hand side is well formed. An effect set is well formed if, for every effect it contains, 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.

[3]Similar to the work by Maffeis et al. [41], we widened the original definition of authority to be about being able to perform any operation on a module, instead of being able to only modify it.

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

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

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

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

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

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

$$\frac{\forall i, j,\ n_i.g_j \in \varepsilon,\ \Gamma \mid \Sigma \vdash n_i : \{\}\ \{y_i \Rightarrow \overline{\sigma_i}\}_s,\ (\texttt{effect}\ g_j \in \overline{\sigma_i} \vee \texttt{effect}\ g_j = \{\varepsilon_j\} \in \overline{\sigma_i})}{\Gamma \mid \Sigma \vdash \varepsilon \ wf} \text{ (WF-EFFECT)}$$

Figure 3.12: Well-formedness rules.

3.4.4 Static Semantics

Wyvern's static semantics is presented in Figure 3.13. 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 and the store typing context Σ, 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. The s subscript on the turnstile in the premise of T-NEW and in the object-declaration typing rules is the same as on the `new` construct and the object types in the syntax and signifies whether we are type checking a pure object or a resource object.

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. Notably, the expressions that are being substituted are always locations, 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

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

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

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

$$\frac{\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma}}{\Gamma \mid \Sigma \vdash e.f : \{\varepsilon\}\ [e/x]\tau}\ \text{(T-FIELD)} \qquad \frac{l : \tau \in \Sigma}{\Gamma \mid \Sigma \vdash l : \{\}\ \tau}\ \text{(T-LOC)}$$

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

$$\frac{\Gamma \mid \Sigma \vdash e : \{\varepsilon_1\}\ \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \mathit{lookup}(\Gamma, \varepsilon_1) = \varepsilon_1' \quad \mathit{lookup}(\Gamma, \varepsilon_2) = \varepsilon_2' \quad \varepsilon_1' \subseteq \varepsilon_2'}{\Gamma \mid \Sigma \vdash e : \{\varepsilon_2\}\ \tau_2}\ \text{(T-SUB)}$$

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

$$\frac{\begin{array}{c}\Gamma_{\texttt{resource}} = \{x : \{x \Rightarrow \overline{\sigma}\}_{\texttt{resource}} \mid x : \{x \Rightarrow \overline{\sigma}\}_{\texttt{resource}} \in \Gamma\} \quad \Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}} \\ \Gamma_{\texttt{pure}},\ y : \tau_1 \mid \Sigma \vdash e : \{\varepsilon_2\}\ \tau_2 \quad \Gamma,\ y : \tau_1 \mid \Sigma \vdash \varepsilon_1\ \mathit{wf} \\ \mathit{lookup}(\Gamma, \varepsilon_1) = \varepsilon_1' \quad \mathit{lookup}(\Gamma, \varepsilon_2) = \varepsilon_2' \quad \varepsilon_1' \supseteq \varepsilon_2'\end{array}}{\Gamma \mid \Sigma \vdash_{\texttt{pure}} \texttt{def}\ m(y : \tau_1) : \{\varepsilon_1\}\ \tau_2 = e\ \ :\ \ \texttt{def}\ m(y : \tau_1) : \{\varepsilon_1\}\ \tau_2}\ \text{(DT-DEFPURE)}$$

$$\frac{\begin{array}{c}\Gamma,\ x : \tau_1 \mid \Sigma \vdash e : \{\varepsilon_2\}\ \tau_2 \quad \Gamma,\ x : \tau_1 \mid \Sigma \vdash \varepsilon_1\ \mathit{wf} \\ \mathit{lookup}(\Gamma, \varepsilon_1) = \varepsilon_1' \quad \mathit{lookup}(\Gamma, \varepsilon_2) = \varepsilon_2' \quad \varepsilon_1' \supseteq \varepsilon_2'\end{array}}{\Gamma \mid \Sigma \vdash_{\texttt{resource}} \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 = e\ \ :\ \ \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2}\ \text{(DT-DEFRESOURCE)}$$

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

$$\frac{\Gamma \mid \Sigma \vdash \varepsilon\ \mathit{wf}}{\Gamma \mid \Sigma \vdash_s \texttt{effect}\ g = \{\varepsilon\}\ \ :\ \ \texttt{effect}\ g = \{\varepsilon\}}\ \text{(DT-EFFECT)}$$

$\boxed{\mu : \Sigma}$

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

Figure 3.13: Wyvern static semantics.

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

$\boxed{lookup(\Gamma, \overline{x.g})}$

$$lookup(\Gamma, \overline{x.g}) = \bigcup_{x.g \in \overline{x.g}} lookup(\Gamma, x.g) \quad \text{(LOOKUP)}$$

$\boxed{lookup(\Gamma, x.g)}$

$$\frac{\Gamma \mid \varnothing \vdash x : \{\}\ \{y \Rightarrow \overline{\sigma}\}_s \quad \texttt{effect}\ g \in \overline{\sigma}}{lookup(\Gamma, x.g) = x.g} \text{ (LOOKUP-STOP)}$$

$$\frac{\Gamma \mid \varnothing \vdash x : \{\}\ \{y \Rightarrow \overline{\sigma}\}_s \quad \texttt{effect}\ g = \{\varepsilon\} \in \overline{\sigma}}{lookup(\Gamma, x.g) = lookup(\Gamma, [x/y]\varepsilon)} \text{ (LOOKUP-RECURSE)}$$

Figure 3.14: Wyvern effect-lookup rules.

the original type, the original type is a subtype of the new type, and when having resolved the effect definitions in the effect sets of both types to the lowest-possible level of effects (the lookup rules will be discussed in Section 3.4.5), the resolved effect set of the original type is a subset of the resolved effect set of the new type.

None of the object declarations produce effects, and so object-declaration type-checking rules do not include an effect set preceding the type annotation. Also, inheriting the distinction from Wyvern's capability-safe module system, when type checking object declarations, we differentiate between pure and resource objects, which is denoted by the subscript under the turnstile of the object-declaration type-checking rules. Then, for declarations, the judgement reads that, in the variable typing context Γ and the store typing context Σ, the declaration d that belongs to a pure or a resource object (the s tag underneath the turnstile) is a well-typed declaration with the type σ.

Correspondingly, there are different rules for a method declaration depending on whether the method declaration is contained in a pure or in a resource object. If a method declaration is contained in a pure object, the method's body must be well typed in a typing context devoid of all resource objects and containing the method argument. The effect set annotating the method must be well formed in the overall typing context extended with the method argument. Furthermore, when all effects are resolved to the lowest-possible level of effects, the effect set annotation the method must be a superset of the effect set the method body actually produced. We use the superset relationship here to impose more limited subtyping. Type checking of a method declaration of a resource object is exactly the same, except for all the checks are done in the overall context without removing anything from it.

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.

Finally, we ensure that the store is well-formed and contains objects that respect their types.

3.4.5 Effect-Lookup Rules

As we already saw in the T-SUB, DT-DEFPURE, and DT-DEFRESOURCE rules above and as we will see more in the upcoming Section 3.4.7, to compare two sets of effects, we use effect-lookup

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

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

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

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

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

Figure 3.15: Wyvern dynamic semantics.

rules, which are presented in Figure 3.14. These rules "look up" the effects following their definitions to the lowest-possible level, at which point we can compare the effect sets canonically. When given a set of effects, we look up each effect in the set separately (LOOKUP). For each effect, we find the type of the variable in the first part of that effect and look for an effect declaration with the name matching the effect name in the second part of the effect after the dot. If we find that the effect is abstract, i.e., the type only declares that effect by does not provide a definition for it, we stop the lookup and return the effect we were looking up last (LOOKUP-STOP). Alternatively, if we find that the effect is concrete, i.e., the type provides a definition for it, we substitute the current object for the variable name in all the effects in the newly found effect definition and recursively call the lookup function on the resulting effect set (LOOKUP-RECURSE).

3.4.6 Dynamic Semantics

The dynamic semantics that we use for Wyvern's effect system is shown in Figure 3.15 and is similar to the one we used in the module system (cf. Figure 2.11) but simpler. 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.

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

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

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

$$\frac{}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i \in 1..n+k}\}_s <: \{x \Rightarrow \sigma_i^{i \in 1..n}\}_s} \text{ (S-WIDTH)}$$

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

$$\frac{}{\Gamma \vdash \{x \Rightarrow \overline{\sigma}\}_{\texttt{pure}} <: \{x \Rightarrow \overline{\sigma}\}_{\texttt{resource}}} \text{ (S-RESOURCE)}$$

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

$$\frac{}{\Gamma \vdash \sigma <: \sigma} \text{ (S-REFL2)}$$

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

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

Figure 3.16: Wyvern subtyping rules.

3.4.7 Subtyping Rules

Wyvern subtyping rules are shown in Figure 3.16. Since, to compare types, we need to compare (i.e., *lookup*) the effects in them, 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. The S-RESOURCE rule was discussed previously in Section 2.5.5 and says that pure objects can be subtypes of resource objects but not other way around. 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: after having been resolved to the lowest-possible level using the lookup rules described above, the effect set of the subtype method declaration must be a subset of the effect set of the supertype method declaration. Finally, an effect definition is trivially a subtype of an effect declaration.

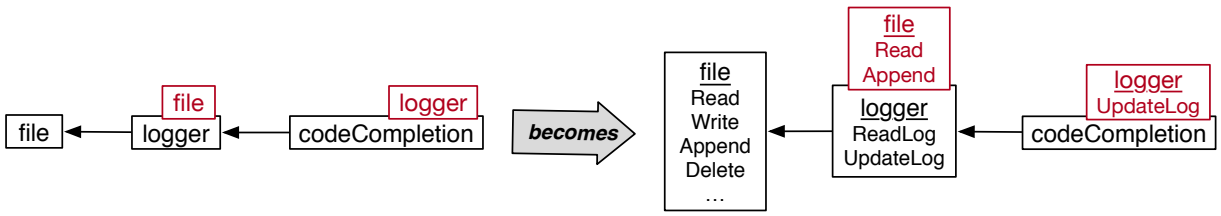


Figure 3.17: Intuition behind using effects to describe module authority Black boxes represent modules, and red boxes represent capabilities. On the left-hand side of the figure is a diagram representing what we knew about resource modules and capabilities in Chapter 2 only relying on our module system design. On the right-hand side of the figure is a diagram augmented with the information about effects that we gain by using Wyvern's effect system.

3.4.8 Type Soundness

The preservation and progress theorems are as follows, and the proofs for both of them are fairly standard and are available in the Appendix B.1.

Theorem (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* $lookup(\Gamma, \varepsilon') \subseteq lookup(\Gamma, \varepsilon)$*, and* $\Gamma \mid \Sigma' \vdash e' : \{\varepsilon'\}\ \tau$*.*

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

3.5 Authority-Related Properties

Our definition of authority, presented in Section 1.1, is based on prior research [24, 48] and says that authority is the ability to operate on resources. Using the extra information that effect members and annotations provide, we can now talk about authority of modules (and objects) in an application.

The intuition behind this enhancement in reasoning about an application's security can be seen in Figure 3.17, which shows the relationship between modules described in Figures 3.1–3.4. Black boxes represent modules, and red boxes represent capabilities. On the left-hand side of the figure is a diagram representing what we knew about resource modules and capabilities in Chapter 2 only relying on our module system design. Namely, the `logger` module has a capability to access the `file` module, and the `codeCompletion` module has a capability to access the `logger` module. On the right-hand side of the figure is a diagram augmented with the information about effects that we gain by using Wyvern's effect system. Namely, now we know that the methods in the `file` module can produce several effects (`Read`, `Write`, etc.) and that the `logger` module uses only two of those effects (the `Read` and `Append` effects). Similarly, the `logger`'s methods can produce two effects (`ReadLog` and `UpdateLog`), but the `codeCompletion` plugin produces only the `UpdateLog` effect on the `logger` module. This allows a security analyst or a software developer to analyze an application security more precisely than if they used only capabilities.

Underpinned by Wyvern's capability safety that is facilitated by the module system, we introduce three authority-related properties of Wyvern: authority safety, authority of an object, and authority attenuation.

3.5.1 Authority Safety

In Section 1.1, we defined an authority-safe programming language as one that provides a way for a software developer to specify and limit modules' (or objects') authority over other modules (or objects) using a set of well-defined rules. Through examples in Sections 3.1–3.3, we illustrated how a software developer could use effect annotations to specify and control modules' authority. Our formal system, described in Section 3.4, ensures that the program behavior adheres to the rules specified by the software developer. Specifically, Wyvern's static semantics (Section 3.4.4) checks that effect annotations correspond to the effects produced by each method body, and the preservation theorem (Section 3.4.8) guarantees that effects produced during execution adhere to the effect annotations in the program. Then, since we proved the type soundness of Wyvern's effect system, we proved Wyvern authority safe.

3.5.2 Authority of an Object

A basic notion in the authority analysis of an application is the notion of an object's authority, which we define next.

Definition 1 (Authority of an object)**.** The authority of an object is a set of effects that the object's methods and fields can produce.

This definition is "outward facing" in a sense that it helps reasoning about authority of objects that use the current object. We chose such definition because it seems to be more useful in a security analysis. For example, if an application's programming interface allows plugins to access a specific module (e.g., the `logger` module described in Figure 3.2), it is useful to be able to determine what effects a plugin could produce by using that module, accessing its fields and calling methods on it.

Formally, we represent an object's authority as a set of $auth$ rules, shown in Figure 3.18. An object's authority (AUTH-OBJECT) is the authority of the object's declarations. Authority of a method declaration (AUTH-DEF) is the effects that the method produces during execution and also the authority of objects of the method's return type. The reason for including the latter authority component is that, whenever the method is called, an object of the return type is returned to and may be operated on by the caller, thus increasing the caller's authority. For the same reason, authority of an object's field (AUTH-VAR) is the authority of objects of the field's type. An effect declaration carries no authority (AUTH-EFFECT).

Authority of objects of a particular type (AUTH-TYPE) is authority of the type's declarations. Authority of a method-declaration type (AUTH-DEFTYPE), a field-declaration type (AUTH-VARTYPE), and a concrete-effect-declaration type (AUTH-CONEFFECTTYPE) is similar to the authority of corresponding declarations in an object. An abstract-effect-declaration type produces no authority (AUTH-ABSEFFECTTYPE).

As an example of how these rules can be applied in practice, if we look at the `logger` module presented in Figure 3.2, using the $auth$ rules, we can determine that `logger` has authority to read

$\boxed{auth(\texttt{new}_s(x \Rightarrow \overline{d}))}$

$$auth(\texttt{new}_s(x \Rightarrow \overline{d})) = \bigcup_{d \in \overline{d}} auth(d) \ \text{(AUTH-OBJECT)}$$

$\boxed{auth(d)}$

$$auth(\texttt{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2 = e) = \varepsilon \cup auth(\tau_2) \ \text{(AUTH-DEF)}$$
$$auth(\texttt{var}\ f : \tau = n) = auth(\tau) \ \text{(AUTH-VAR)}$$
$$auth(\texttt{effect}\ g = \{\varepsilon\}) = \varnothing \ \text{(AUTH-EFFECT)}$$

$\boxed{auth(\tau)}$

$$\frac{\tau = \{x \Rightarrow \overline{\sigma}\}_s}{auth(\tau) = \bigcup_{\sigma \in \overline{\sigma}} auth(\sigma)} \ \text{(AUTH-TYPE)}$$

$\boxed{auth(\sigma)}$

$$auth(\texttt{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2) = \varepsilon \cup auth(\tau_2) \ \text{(AUTH-DEFTYPE)}$$
$$auth(\texttt{var}\ f : \tau) = auth(\tau) \ \text{(AUTH-VARTYPE)}$$
$$auth(\texttt{effect}\ g) = \varnothing \ \text{(AUTH-ABSEFFECTTYPE)}$$
$$auth(\texttt{effect}\ g = \{\varepsilon\}) = \varnothing \ \text{(AUTH-CONEFFECTTYPE)}$$

Figure 3.18: Rules defining authority of an object.

the log file (the `ReadLog` effect) and to update the log file (the `UpdateLog` effect), but no other authority. Although this example may seem trivial, knowing an object's authority is useful if we analyze objects that are more complex than the `logger` object and also for object comparison, like we demonstrate in the next section.

3.5.3 Authority Attenuation

Introduced in Mark Miller's dissertation [48], the notion of authority attenuation can be described as follows. If a module (or an object) accesses a resource and produces less than the total possible set of operations on that resource, we say that that module (or object) *attenuates* the resource. For example, consider the modules presented in Figure 3.17 (the code for which is shown in Figures 3.2–3.4). We observe that, while the `file` module can have a number of effects (`Read`, `Write`, `Append`, etc.), the `logger` module produces only two of `file`'s effects (`Read` and `Append`). Then, any module that uses `logger` and does not have access to the `file` module (e.g., the `codeCompletion` plugin module) can produce on `file` *at most* the two effects `logger` can produce. Thus, the `logger` module attenuates the `file` capability by giving access to only a subset of `file`'s effects.

To aid a security analyst in a formal security analysis of an application, we formalized the notion of authority attenuation. Relying on our effect system, our definition of authority attenuation is static. We only examine an object's code and do not know which specific objects the object uses at run time. Instead, we can talk about objects of a specific *type* that the object uses. Our definition of authority attenuation benefits from this since we can talk about *groups of objects* any object in which is attenuated. For example, using our static definition of authority attenuation, instead of knowing that the `logger` module attenuates the `file` module (which is of type `File`),

$\boxed{tLookup(\Gamma, \tau, \overline{x.g})}$

$$tLookup(\Gamma, \tau, \overline{x.g}) = \bigcup_{x.g \in \overline{x.g}} tLookup(\Gamma, \tau, x.g) \quad \text{(tLookup)}$$

$\boxed{tLookup(\Gamma, \tau, x.g)}$

$$\frac{\Gamma \mid \varnothing \vdash x : \{\}\ \tau}{tLookup(\Gamma, \tau, x.g) = \tau.g} \text{ (tLookup-Stop)}$$

$$\frac{\Gamma \mid \varnothing \vdash x : \{\}\ \tau' \quad \tau' \neq \tau \quad \tau' = \{y \Rightarrow \overline{\sigma}\}_s \quad \texttt{effect}\ g \in \overline{\sigma}}{tLookup(\Gamma, \tau, x.g) = \tau'.g} \text{ (tLookup-Stop2)}$$

$$\frac{\Gamma \mid \varnothing \vdash x : \{\}\ \{y \Rightarrow \overline{\sigma}\}_s \quad \tau \neq \{y \Rightarrow \overline{\sigma}\}_s \quad \texttt{effect}\ g = \{\varepsilon\} \in \overline{\sigma}}{tLookup(\Gamma, \tau, x.g) = tLookup(\Gamma, \tau, [x/y]\varepsilon)} \text{ (tLookup-Recurse)}$$

Figure 3.19: Wyvern effect-lookup rules that target a specific type.

we know that `logger` attenuates *all* objects of type `File`.

In essence, our formal definition says that if we let F_1 be the set of effects that represents an object's authority and F_2 be the set of effects that represents authority of objects of a specific type. Then, if F_1 and F_2 share at least one effect and there is at least one effect that is in F_2 but not in F_1, we say that the object attenuates objects of that type. For example, if we let F_1 be the set of effects that represents the `logger`'s authority and F_2 be the set of effects that represents authority of objects of type `File`. Then, if F_1 and F_2 share at least one effect and there is at least one effect that is in F_2 but not in F_1, we say that `logger` attenuates objects of type `File`. Formally, we write these conditions as follows.

Definition 2 (Authority Attenuation)**.** An object o attenuates objects of type τ, if

1. $F_1 = tLookup(\Gamma, \tau, auth(o))$, $F_2 = tLookup(\Gamma, \tau, auth(\tau))$,
2. $F_1 \cap F_2 \neq \varnothing$, and
3. $F_2 \setminus F_1 \neq \varnothing$.

First, using the $auth$ rules (shown in Figure 3.18), we find authority of object o and of objects of type τ. Then, we use the $tLookup$ rules, which are shown in Figure 3.19, to "normalize" the two effect sets, thus making it possible to compare them. Finally, we compare the two effect sets.

The $tLookup$ rules serve to support the static nature of our definition of authority attenuation. Similarly to the $lookup$ rules (Figure 3.14), $tLookup$ rules resolve effects to lower-level effects. However, they differ in that the $tLookup$ rules "search" for effects of an object of a particular *type* and stop when an object of that type is found.

When we apply $tLookup$ to a set of effects, we apply $tLookup$ to each effect in that set (tLookup). If the type that we are looking for is the type of the current object (tLookup-Stop), we return the "normalized" form of the effect, which differs from the original form in that we substitute the variable name with the type name. If we encounter an abstract effect (tLookup-Stop2), we return the "normalized" form of that effect that uses the type of the current object. Otherwise, the effect is concrete, and we proceed by examining the effect's

definition (TLOOKUP-RECURSE).

As an example, let us apply our definition of authority attenuation to the `logger` module and the objects of type `File` (e.g., the `file` module) from Figure 3.17. Using the $auth$ and $tLookup$ rules on the `logger` object, we determine that `logger`'s authority is:

$$\begin{aligned}
F_1 &= tLookup(\Gamma, File, auth(logger)) \\
&= tLookup(\Gamma, File, this.ReadLog, this.UpdateLog) && (this \text{ is implicit}) \\
&= tLookup(\Gamma, File, this.ReadLog) \cup tLookup(\Gamma, File, this.UpdateLog) \\
&= tLookup(\Gamma, File, f.Read) \cup tLookup(\Gamma, File, f.Append) \\
&= \{File.Read, File.Append\}
\end{aligned}$$

Similarly, we determine that the authority of objects of type `File` is:

$$\begin{aligned}
F_2 &= tLookup(\Gamma, File, auth(File)) \\
&= tLookup(\Gamma, File, this.Read, this.Write, this.Append, this.Delete, \ldots) && (this \text{ is implicit}) \\
&= tLookup(\Gamma, File, this.Read) \cup tLookup(\Gamma, File, this.Write) \\
&\cup tLookup(\Gamma, File, this.Append) \cup tLookup(\Gamma, File, this.Delete) \cup \ldots \\
&= \{File.Read, File.Write, File.Append, File.Delete, \ldots\}
\end{aligned}$$

Then, comparing the two sets, we have:

$$\begin{aligned}
F_1 \cap F_2 &= \{File.Read, File.Append\} \neq \varnothing \\
F_2 \setminus F_1 &= \{File.Write, File.Delete, \ldots\} \neq \varnothing
\end{aligned}$$

Therefore, by our definition, the `logger` module attenuates modules of type `File`.

It is possible to create a more general formal definition of authority attenuation by, instead of considering one object that attenuates objects of a specific type, considering objects of one type that attenuate objects of another type. This version of the authority attenuation definition is presented in Appendix B.2.

3.6 Implementation

Effect checking was implemented as part of the open-source Wyvern compiler and interpreter, available on GitHub: `https://github.com/wyvernlang/wyvern`, and is tested by the `wyvern.tools.tests.EffectSystemTests` test suite.

3.7 Limitations

A frequently cited limitation of effect systems is the high annotation burden. While outside the scope of this dissertation, recent work by Craig et al. [14] demonstrated how, in the presence of first-class modules acting as object capabilities, effect annotations can be inferred at a more granular level. This work showed that capabilities allow the effect information to be inferred "for free" by building on the absence of globally accessible objects.

Common examples of effect systems in the real world are exceptions in languages such as Java, C#, and Scala [29]. Keeping track of all possible effects in the code is a nontrivial problem, even in the case of checked exceptions, and can be addressed by polymorphic effects [64] or effect aggregation as we described above. To address these issues, Lubin [36] extended the work of Craig et al. [14] on Wyvern effects to account for mutable state and effect polymorphism, demonstrating how effects with first-class modules can be used by programmers with only the essential effect annotations. When combined with the work on abstract effects as presented here, we believe our approach achieves both high assurance and high programming language usability.

3.8 Related work

Origins of Effect Systems. Effect Systems were originally proposed by Lucassen [37] in 1987 to track reads and writes to memory. Lucassen and Gifford extended this effect system to support polymorphism the following year [38]. Effects have since been used for a wide variety of purposes, including exceptions in Java and asynchronous event handling [11]. Turbak previously proposed effects as a mechanism for reasoning about security [71], which is the main application that we discuss.

Algebraic Effects, Generativity, and Abstraction. Algebraic effects and handlers [59, 60] are a way of implementing certain kinds of side effects such as exceptions and mutable state in an otherwise purely functional setting. Bračevac et al. [11] 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. Their OCaml implementation builds on Multicore OCaml [18], and they observe that declaring an effect type in a module signature is a way of getting per-module generativity. They do not explore abstract effects, per-object generativity (e.g., as in different effects for every `File` object instance), nor do they formalize generative effects.

Zhang et al. [76] 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.

Biernacki et al. [6] 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 3.3, since some of our scenarios rely critically on abstracting lower-level events as higher-level ones.

JML's data groups [33] have several similarities to Wyvern's effect system, including declarations that can have custom identifiers, definitions that can be defined using a collection of field locations and other data groups, and abstraction that allows reasoning about effects in client code without knowing the underlying definitions. However, data groups are used to abstract over only a specific type of effects, namely, modifications to heap locations, whereas Wyvern effects can represent more general notion of effects. Also, Wyvern's effect system provides the ability to either keep an effect abstract or make its definition public, and Wyvern's effects depend on individual objects, allowing for more flexibility and more precise effect analysis, respectively.

Notable Effect-System Implementations. Several programming languages and language extensions provide general effect systems. Scala has two effect-related extensions: one is a generic effect-checking framework that supports effect polymorphism [64], and the other specifically supports algebraic effects [9]. IDRIS [10] supports an embedded domain specific language that, relying on the host language being dependently typed, allows capturing and handling algebraic effects. Koka [32] requires effects to be an explicit part of the type signature of a function, implements polymorphic effects using row polymorphism, and allows for effect inference. Eff [5] supports first-class effects and handlers, and takes the algebraic approach to computational effects, viewing them as algebraic operations. Finally, Frank [34] introduced a form of effect polymorphism that avoids mentioning effect variables in the source code, instead relying on the fact that an operator's effects are always instantiated with all algebraic effects permitted by the current typing context.

Similar to our design, languages that support polymorphic effects (Koka, Frank, and Rytz et al.'s extension of Scala) allow flexibility of implementation in terms of what lower-level effects are used to implement a function. However, none of these effect systems feature effect abstraction: allowing a module to declare an effect that is implemented in terms of other effects, but hiding what effects those are from clients.

Marino and Millstein [42] discuss an effect system in which application-specific effects can be defined. One of their examples is system calls that can block. Our focus on system resources captures a slightly different set of system calls, namely, those that have an impact on an external resource.

Reasoning About Authority of Code. Object capabilities [48] have been used to reason about the authority of an object to access a resource in the system, based on whether that object possesses an appropriate capability (typically an object pointer). This was first formalized by Maffeis et al. [41], who used a topology criterion to reason about what resources a module can affect. The idea is that if a resource such as a file is reachable from some object, then that object has authority over the resource. However, this criterion is an approximation: perhaps the file is transitively reachable, but due to information hiding the file may not actually be directly accessible (e.g., the resource is stored in a private field of some intermediate object). Our analysis of authority is more precise in two ways. First, effects distinguish different classes of operations, such as read, write, and append, and they can characterize not just what resource an object has authority to access but what operations it has authority to perform on those resources. Second, since an intermediate object may only use a resource in certain ways, we can reason about

authority attenuation, which is impossible in a definition based purely on transitive topology.

While Maffeis et al. used definitions based on the run-time semantics of a language, more recent work attempts to support static reasoning about authority. One solution is to use information flow policies to reason about the capability propagation directly [17], which comes at a cost of defining and maintaining such system-wide policies. Another solution, proposed by Devriese et al. [16], characterizes effects as a set of side-effecting commands, then applies logical relations and parametric reasoning to analyze the possible effects of untrusted code. Devriese et al.'s approach is precise but requires heavyweight mathematical machinery. In contrast, our goal is to retain as much precision as possible in the simpler setting of an effect system.

Authority Attenuation. Although a number of works on object capabilities and authority safety mention and explain authority attenuation (e.g., [45, 48]), the only work on formalizing authority attenuation that we are aware of is a recent workshop presentation by Loh and Drossopoulou [35]. In the presentation, the authors used Hoare triples and invariants to show how authority can be attenuated in a restricted document object model (DOM) tree. In contrast, our approach to authority attenuation uses effect abstraction and is more generally applicable, being able to describe a wide range of situations.

Chapter 4

Evaluation

As an evaluation of our design of Wyvern's module system and effect system, we present ways to mitigate the threats and attack scenarios described in the threat model (Section 1.2), as well as a case study of an extensible text-editor application written in Wyvern, on which we performed a security analysis.

4.1 Threat Mitigation

Conventional programming languages typically allow any module to access any resource and permit passing pointers to those resources in difficult-to-track ways. To ensure that the principle of least privilege is obeyed in a software application, such languages usually use one of the following two ways. They require either sandboxing some parts of the application code, which adds computational overhead and is complicated and error-prone [13, 39], or a manual code inspection, which is error-prone, laborious, and time-consuming.

In contrast, Wyvern simplifies the task of performing a security audit by requiring software analysts to examine, in many cases, only modules' interfaces and not their code. Specifically, security analysts need to inspect what capabilities a module receives on its instantiation or as method arguments, and methods' effect annotations, i.e., what effects the module's methods produce on the capabilities that the module uses. Once the resource access and use is specified in module interfaces, Wyvern's type-and-effect system ensures at compile time that the specification is enforced. If a security analyst discovers an instance of privilege abuse during the security audit, Wyvern provides a mechanism to preclude this. For example, the security analyst may modify the code to not pass the capability to the module at its instantiation or provide an attenuated version of the capability that either makes the method "harmless" or removes it, thereby disabling the third party's ability to call the abusive method.

Both potential attacks described in the threat model (Section 1.2) can be mitigated by examining modules' interfaces. In many cases, an attempt by a third-party module to exploit the application by itself is obvious from looking at the module's interface. To gain access to system resources or native code, a module must receive an appropriate capability either on its creation or as a method argument. Then, if a third-party module tries to gain unauthorized access to a system resource, it is evident from examining the module's interface and checking what capabilities it

receives via these two mechanisms. Importantly, to be able to use a native module (e.g., Python's `builtins` module), a third-party module needs to receive an appropriate capability. In addition, operations on system resources are specified in effect annotations on each of the module's methods. Then, if a third-party module tries to perform an unauthorized operation on a system resource necessary to implement the primary functionality of the malicious code, it is evident from examining the module's interface and checking the effect annotations on its methods. If a third-party module requires unauthorized capabilities or performs unauthorized operations on the capabilities that are to be passed into it, the developer or security analyst may choose not to instantiate that module or not to call the abusive method.

To protect against the attacks described in the threat model (Section 1.2) where third-party code exploits the application via fallible application code, in many cases, it is sufficient for security analysts to examine the interfaces of the modules that are used by third-party modules. In particular, security analysts must ensure that application modules:

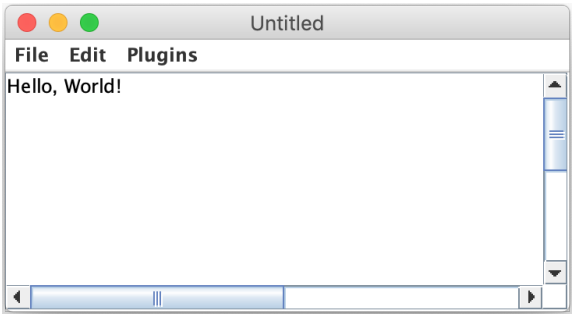
1. Properly manage the resources they need to implement their own functionality by:
   (a) not requiring capabilities unnecessary for implementing their functionality;
   (b) not unnecessarily exposing system resources (e.g., by returning a reference to a system resource from a method); and
   (c) not performing operations on system resources that are not necessary to implement their functionality.
2. If application modules are used to perform operations on system resources on their clients' behalf, their clients are required to pass in an appropriate capability to access each system resource.

If any of these violations are found, the security analyst should modify modules' interfaces to eliminate unnecessary privilege and to remove the unnecessary resource exposure, requiring the software developer to change the modules' code to obey the new interfaces.

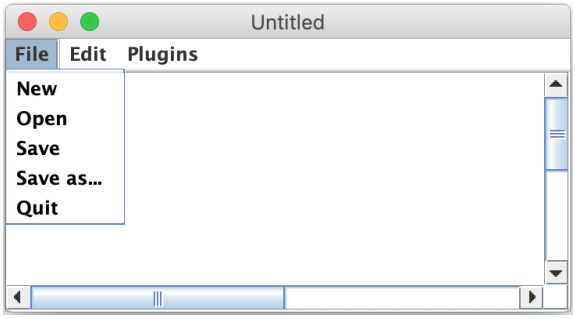
As a limitation of our approach, if the desired security properties cannot be expressed in the module interfaces in terms of required capabilities and produced effects, the security analyst needs to fall back to a conventional code-inspection-based strategy. For example, if the abusive module legitimately requires the privilege that it is granted (i.e., the module legitimately requires access to a resource, and it is legitimate for the module to perform all operations specified by effect annotations on its methods) and misuses it, to detect the violation, the security analyst must resort to inspecting the code of the abusive module.

4.2 Case Study: An Extensible Text-Editor Application

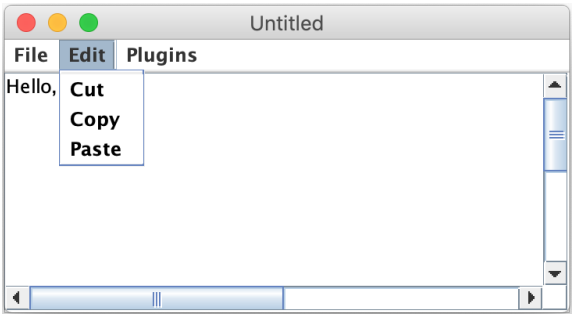
To evaluate our module-system and effect-system designs more directly, we used Wyvern to create an extensible text-editor application. In this section, we describe the text-editor application and plugins for it that we implemented, present a security analysis of our implementation, and state and discuss the observations that we made during the implementation process.



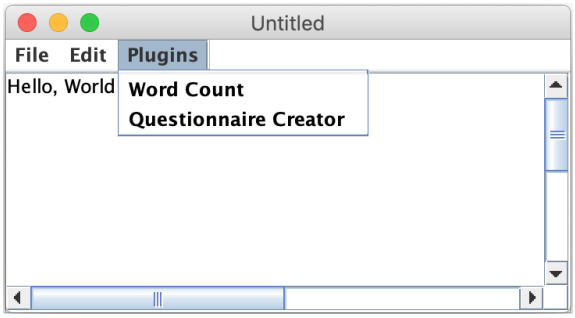
(a) Text area.



(b) The "File" menu.



(c) The "Edit" menu.



(d) The "Plugins" menu.

Figure 4.1: Screenshots of the text-editor application.

4.2.1 Application Description

We implemented a text-editor application[1] that provides the basic text-editing functionality. When started, the text-editor window has a text area where the user may enter or edit text (Figure 4.4a). The title bar shows the path to the currently opened document or "Untitled" if the document have not been saved yet. The menu bar has three options and allows users to perform operations on files (Figure 4.4b), perform editing operations on the text in the text area (Figure 4.1c), or run plugins (Figure 4.1d).

Conceptually, we divide the implementation into two parts: the text-editor part, which consists of 233 lines of code, and the plugins part, which consists of 110 lines of code.

Program Structure

The program starts by compiling and running the `main.wyv` file, which contains the top-level script that instantiates the text-editor application. The main application module is called `textEditor`, and it is of type `TextEditor`.

To build the application, we used Wyvern's Java backend. We created a Java class[2] that provides access to the Java objects necessary for the text editor's execution. On the Wyvern side,

[1]The code for the text-editor application and its plugins is available online: `https://github.com/wyvernlang/wyvern/tree/master/examples/text-editor`

[2]`https://github.com/wyvernlang/wyvern/blob/master/tools/src/wyvern/stdlib/support/TextEditorHelper.java`

```
resource type Plugin
  effect Run
  def getName(): {} String
  def run(): {Run} Unit
```

Figure 4.2: The `Plugin` type that each text editor's plugin must implement.

we created types which correspond to the Java objects that are used by the text editor's plugins, namely, `OptionPane`, `TextArea`, and `UIManager`.

The text-editor application maintains a log of every event that happens inside it. This logging functionality is implemented in the `logger` module, which is of type `Logger`.

Finally, the application is extensible and allows third-party plugins. All plugins must implement the `Plugin` type, which is shown in Figure 4.2. Since the application policy is that *all* events must be recorded in the log, the text editor's plugins must use the `logger` module to update the log file as needed.

Adding a Plugin

All the plugin files must be in the `plugins` directory. There are three steps to add a plugin to the text-editor application. Each plugin must be:

1) imported,
2) instantiated with the resources necessary for its operation (and no more), and
3) registered to be displayed in the "Plugins" menu or set to run (once) when the application starts.

These steps require modifying the `textEditor` module. The plugin imports must happen in the beginning of the `textEditor` module. The plugin instantiation happens after the text editor is fully instantiated (except for the plugins) because plugins may need some elements of the text editor for their operation (e.g., the text area). After plugins are instantiated, they are either registered with the menu or run.

Plugins registered with the "Plugins" menu are those that depend on the user's input. Registering a plugin with the menu enables the user to activate the plugin on demand during the text editor's execution. For example, a plugin that counts the number of words in the currently opened document must capture the latest edits that the user made to the document, and so that plugin should be registered with the menu. In contrast, plugins that are run during the text editor's setup are those that set some of the text editor's configurations. For example, a plugin that sets the text editor's theme should be run during the text editor's setup.

Plugins

We implemented three plugins.[3].

[3]In fact, we implemented four plugins, and there are four plugins in the online code repository. The extra plugin is the `lightTheme` plugin which sets the theme of the text editor to be like in Figure 4.1. However, we removed this plugin from our analysis here because its implementation is similar to the `darkTheme` plugin.

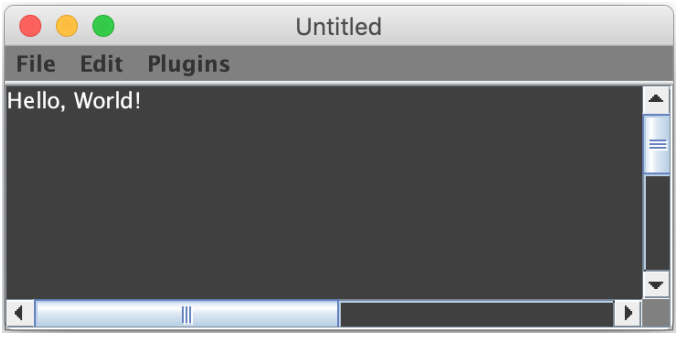
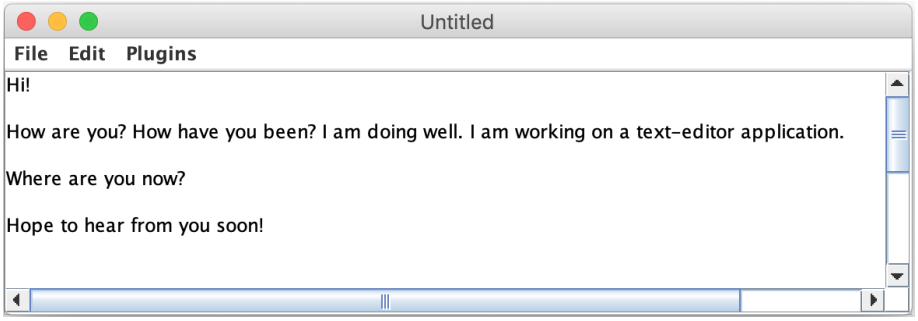
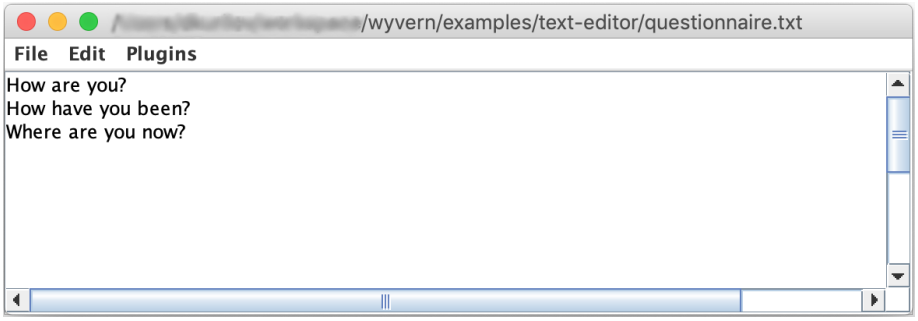


Figure 4.3: Text editor in the dark theme provided by the `darkTheme` plugin.



(a) The original document.



(b) The questionnaire created from the original document.

Figure 4.4: Text editor's `questionnaireCreator` plugin in action.

1. The `darkTheme` plugin sets the theme of the text editor to have a dark background and light text (Figure 4.3).
2. The `questionnaireCreator` plugin extracts questions from the currently opened document and creates a questionnaire in a separate file (Figure 4.4).
3. The `wordCount` plugin counts the number of words in the currently opened document and displays that number to the user in a pop-up window (Figure 4.5).

The `darkTheme` plugin is run only once during the text editor's setup, whereas the `questionnaireCreator` and `wordCount` plugins are registered with the "Plugins" menu, and the user may run them at any time and as many times as they want.

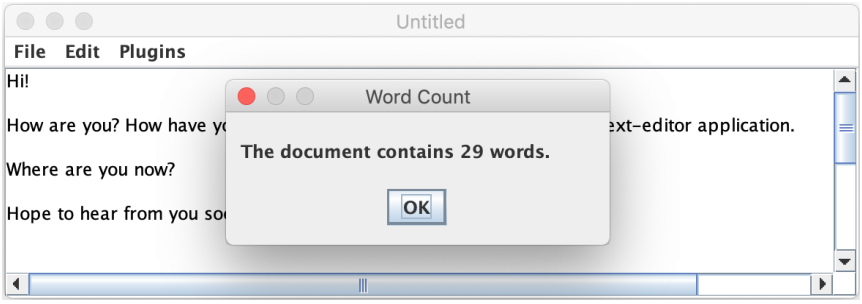


Figure 4.5: Text editor's `wordCount` plugin in action.

Implementation Component Summary

Overall, the implementation consists of the following thirteen components:

- six types:
  - three types that describe modules of the text editor: `TextEditor`, `Logger`, and `Plugin`;
  - three types that describe Wyvern-attenuated Java objects that are used to implement text editor's user interface and are used by the text editor's plugins: `OptionPane`, `TextArea`, and `UIManager`;
- one pure module that defines a set of globally available, user-interface effects: `uiEffects`;
- two resource modules implementing the main text-editor functionality: `textEditor` and `logger`;
- one top-level script that distributes the access to the main system resources and starts the application: `main`;
- three resource[4] modules that implement the three plugins: `darkTheme`, `questionnaireCreator`, and `wordCount`.

4.2.2 Security Analysis

The main goal for designing our module system and effect system is to make it easier for a software developer to follow the principle of least privilege and for a security analyst to verify that the principle is indeed followed. Specifically, we designed the two systems so that:

1. It is possible for a software developer to express the restrictions that they want to impose on the access to and the use of resources, and
2. It is easy for a security analyst to verify what resources are accessed and how they are used.

Procedure

We rely on Wyvern's capability safety and type soundness, which implies authority safety, to ensure that the important information about module accesses and uses is obvious from looking only at modules' interfaces and not their code. Our formal system and Wyvern's compiler, in

[4]All the plugin modules are resource modules because they access user-interface-related resource modules.

which we implemented our formal system, guarantee that fields and method bodies do not allow any extra privilege,[5] thus simplifying the task of following the principle of least privilege in a Wyvern application. To this end, to analyze the text-editor application, we removed all of its code, except for:

- module headers,
- type definitions, and
- effect declarations.

We used the remaining code to establish the privilege that the text editor's modules have.

Using the reduced version of the text editor's code, we determined the modules' accesses and authority as shown in Figure 4.6. In the diagram, the boxes denote modules, and the arrows denote module accesses. If an arrow goes from module A to module B, A accesses B. Modules that are relevant for our discussion about code privilege are augmented with the information about their authority, i.e., effects. For each such module, under its name is a list of effects it declares. Red boxes on top of module boxes are capabilities the modules have along with lists of effects the modules produce on those capabilities.

As an example of how we constructed the diagram in Figure 4.6, consider the reduced version of the `logger` module's code shown in Figure 4.7. From the module header (line 5), we see that `logger` accesses the `logFile` module, an instance of which must be passed in on `logger`'s instantiation. Thus, there is an arrow from `logger` to `logFile` in the diagram, and also the `logFile` capability is in the red box above the `logger`'s box. The parameter to the only method `logger` has is not capability-protected (line 3). The `logger` module declares an `Update` effect (line 6), which is defined to be `logFile.Append`. Thus, under the `logger`'s name is the `Update` effect, and in the red box on top of the `logger`'s box, `logFile` has the `Append` effect. Importantly, the `logger` module accesses no other modules and declares no other effects, which is consistent with the diagram. We performed a similar type of reasoning for each module in the text-editor application.

During the implementation of the text-editor application, for the user-interface-related (UI-related) resources, namely, `uiManager`, `textArea`, and `optionPane`, we maximally limited their exposure to plugins. We wrapped the original UI objects into objects that provide only the methods that plugins call and, correspondingly, have only effects that those methods produce. Therefore, each UI-related effect is produced by at least one plugin.

In addition, to define the effects of the UI-related modules, we made a design choice to use globally available effects (discussed in detail in Section 3.3.3). Thus, we defined the UI-related globally available effects in the pure module called `uiEffects`. We also used globally available effects, instead of the UI modules' own effects, in the plugins.

To determine the effects that the plugins produce on the UI modules, we used a correspondence of the globally available effects with the UI modules' own effects. For example, consider the code snippets of the `OptionPane` type and the `wordCount` plugin shown in Figure 4.8. When run, one of the effects that the `wordCount` plugin produces is the `uiEffects.ShowDialog` effect (line 14). Examining the resources that `wordCount` has access to (lines 8–10), we find `optionPane` of type `OptionPane`. Looking at the `OptionPane`'s effects, we see that its `ShowDialog` effect is defined to be `{uiEffects.ShowDialog}`. Thus, we conclude that the `wordCount` plugin

[5]Currently except for the cases when underlying effects are the same, as was mentioned in Section 3.3.3.

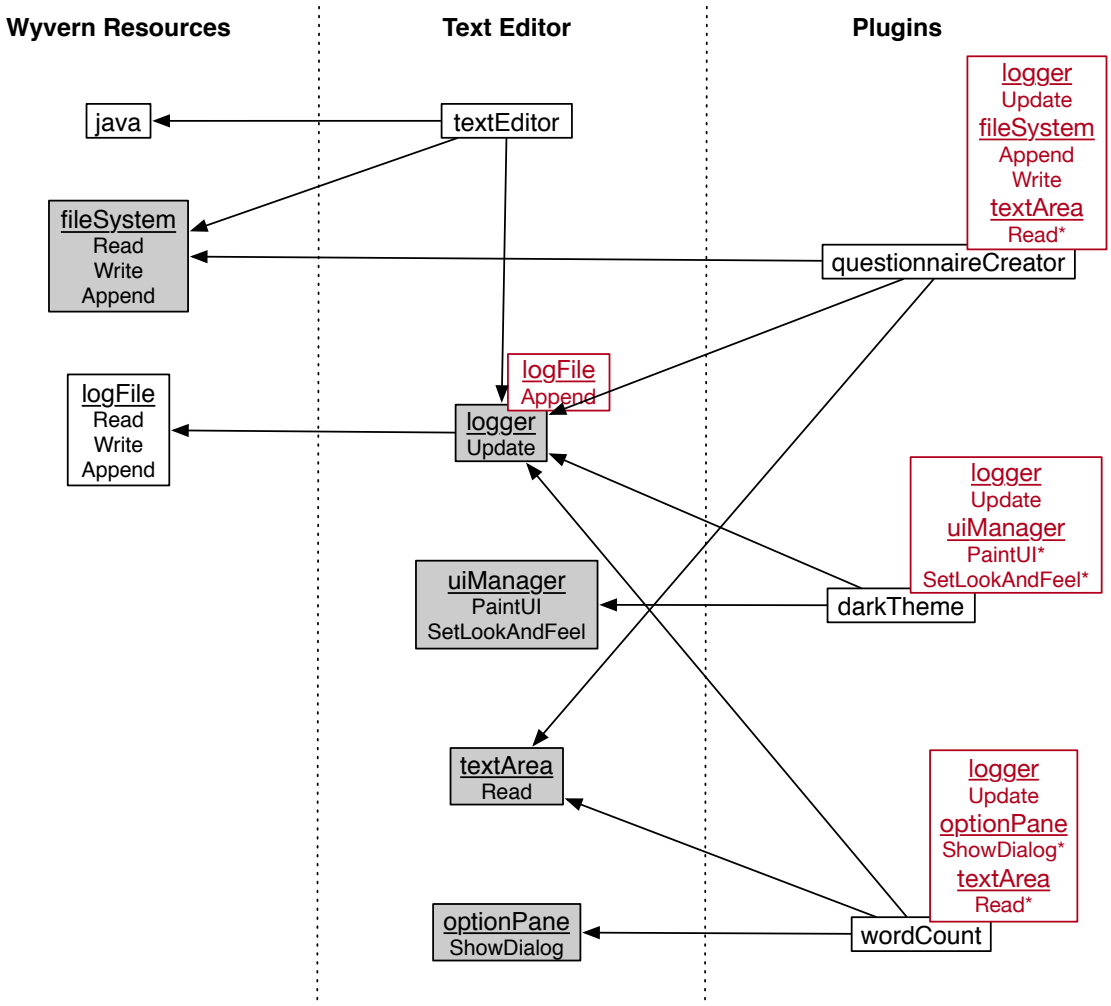


Figure 4.6: Module accesses and authority in the text-editor application deduced from examining only modules' interfaces and types. The boxes denote modules, and the arrows denote module accesses. If an arrow goes from module A to module B, A accesses B. Modules that are relevant for our discussion about code privilege are augmented with the information about their authority, i.e., effects. For each such module, under its name is a list of effects it declares. Red boxes on top of module boxes are capabilities the modules have, along with lists of effects the modules produce on those capabilities. Gray boxes represent modules that are directly accessed by plugin modules. Effects marked with an asterisk were determined indirectly, using the correspondence between the effects of UI-related objects and globally available UI-related effects.

produces the `ShowDialog` effect on the `optionPane` module. In a similar way, we were able to unequivocally match all the UI-related effects. In Figure 4.6, the UI-related effects that we deduced using this method are marked with an asterisk.

```
resource type Logger
  effect Update
  def updateLog(msg: String): {Update} Unit

module def logger(logFile: fileSystem.File): {} Logger
effect Update = {logFile.Append}
```

Figure 4.7: The reduced version of code pertaining to the `logger` module.

```
import uiEffects
resource type OptionPane
  effect ShowDialog = {uiEffects.ShowDialog}
  def showMessageDialog(message: String, title: String, messageType: Int):
                                                          {ShowDialog} Unit
  def getPlainMessageValue(): {} Int

module def wordCount(logger: Logger,
                     textArea: TextArea,
                     optionPane: OptionPane):
                     {} Plugin[{logger.Update,
                                uiEffects.ReadTextArea,
                                uiEffects.ShowDialog}]
effect Run = {logger.Update, uiEffects.ReadTextArea, uiEffects.ShowDialog}
...
```

Figure 4.8: Code snippets of the `OptionPane` type and the `wordCount` plugin.

Analysis

From the security perspective, it is important to inspect the boundary between the trusted, in-house code and the untrusted, third-party code, and verify that the interactions between the two follow the principle of least privilege. In the text-editor application, plugins may be written by some third party, and thus their code is untrusted. Then the boundary between the trusted and untrusted code consists of the modules that the plugins access. In Figure 4.6, such modules are represented by gray boxes.

The boundary between the trusted and untrusted code must be inspected in terms of three aspects:

1. What modules are requested to be passed in to instantiate third-party modules,
2. Whether the boundary in-house modules expose other in-house modules, and
3. How the boundary in-house modules are used by third-party modules, i.e., what operations are performed on the boundary in-house modules by third-party modules.

The first two aspects rely on capability safety. The first one is evident from examining plugins' module headers, and the second one is evident from examining whether the boundary in-house modules' methods return capabilities to access other modules. The last aspect relies on authority safety and is captured via effect annotations on the third-party modules' methods.

The information necessary for verifying all three aspects is shown in the diagram in Fig-

ure 4.6. The information required to verify the first two aspects is reflected in the arrows between the modules, and to decide whether the principle of least privilege is followed, we need to check the logic of whether the existing arrows coming from the plugin boxes are necessary. For example, the `questionnaireCreator` plugin has access to `fileSystem`, `logger`, and `textArea`, which is shown in the diagram as three arrows going from the `questionnaireCreator` box to the three boxes corresponding to the in-house modules. Considering the functionality that the `questionnaireCreator` plugin provides, all three accesses are required: the plugin needs access to `fileSystem` to create a file containing the resulting questionnaire, to `logger` to obey the text-editor's application policy to log all the actions that take place inside the application, and to `textArea` to read the current version of the opened document's text to process into a questionnaire. If the plugin was able to access any other modules, which would have been reflected in the diagram as more arrows going from the `questionnaireCreator` box, those accesses would have been unauthorized. Thus, the `questionnaireCreator` plugin has just enough capabilities to implement its functionality and no more.

The information required to verify the third aspect is shown in the red boxes above each plugin box, and to decide whether the principle of least privilege is followed, we need to check the logic of whether the effects produced on each capability are authorized. For example, the `questionnaireCreator` plugin produced the `Update` effect on `logger`, the `Append` and `Write` effects on `fileSystem`, and the `Read` effect on `textArea`. Considering the functionality that the `questionnaireCreator` plugin provides, all effects are required: the plugin produces the `Update` effect on `logger` to update the log file; the `Write` and `Append` effects on `fileSystem` to create a new file containing the resulting questionnaire and to append to it every time a question is encountered in the original text, respectively; and the `Read` effect on `textArea` to read the current version of the opened document's text to process into a questionnaire. If the `questionnaireCreator` plugin had any more effects, which would have been reflected in the red box above the `questionnaireCreator` box, those effects would have been unauthorized. Thus, all the effects that the `questionnaireCreator` plugin produces are authorized, and the plugin has just enough authority to implement its functionality and no more.

In a similar way, one can verify the three security aspects for all the boundary and plugin modules, and conclude that:

1. Only modules necessary for the plugins' functionality are passed in to instantiate them,
2. The boundary in-house modules expose no extra in-house modules, and
3. All the plugins use the boundary in-house modules in an authorized way.

Thus, the implementation of the text-editor application follows the principle of least privilege, and all the plugins are given just enough privilege to implement their functionality and nothing more. They are given access to the minimal number of resources and, on those resources, they perform only the necessary operations.

Conclusion

The privilege-analysis results show that our designs of the module and effect systems helped enforcing the desired security guarantees. It is possible to express the limitations on resource access and use needed to follow the principle of least privilege. Our designs also simplify verifying what resources are accessed and how they are used by making the security-related information

more obvious and reducing the amount of code one needs to inspect to only the module interfaces and not their code.

4.2.3 Observations and Discussion

During the implementation and analysis of the text-editor application, we made several observations that stem from the way we designed Wyvern's module and effect systems. Next, we present and discuss the benefits and limitations of our approach that we observed.

Benefits

We observed the following six benefits of our module-system and effect-system designs.

Simplified Code Inspection. One of the core advantages of using Wyvern is that, due to formal guarantees that Wyvern provides, during a security analysis, a security analyst needs to inspect less code than in other programming languages. Specifically, to deduce what resources each application module accesses and how those resources are used, one needs to examine only modules' interfaces and not their code.

As described in Section 4.2.2, to perform a security analysis of the text-editor application and its plugins, we removed all the application code, except for the interfaces, namely, module headers, type definitions, and effect declarations. In our version of the text-editor implementation, the interfaces accounted for 69 lines (out of 343 lines in total) or 20% of code. This is a significant code reduction, as we reduced the amount of code to a fifth of the original. However, as the text-editor application grows in size and more plugins are added, we expect to be able to remove an even larger fraction of code. The reason for that is that, in large software systems, more methods tend to be private methods and thus are not part of the interface of a module, and their signatures need not be inspected.

Thus, our approach proved to reduce the amount of code a security analyst needs to examine to evaluate an application's security, and we expect an even greater reduction with the growth of the code base.

Information Hiding and Polymorphism. Our module-system and effect-system designs facilitate the principle of information hiding, i.e., they allow flexibility in what resources are used and how they are used in modules that implement the same type. To achieve this, three features of our designs come together: type abstraction in terms of what resources must be passed into a module on its instantiation, effect abstraction, and effect polymorphism.

There are two examples that we observed in the text-editor application. The first example lies in the plugin modules. For their (legitimate) functionality, plugins may use any available resource and also may use same resources in different ways, while implementing the same type. For example, consider the `darkTheme` and `questionnaireCreator` plugin modules, relevant code snippets for which are presented in Figure 4.9. Both plugin modules implement the `Plugin` type (shown in Figure 4.2). However, as can be seen from lines 2 and 8 in the code snippets, except for the logging module, `darkTheme` and `questionnaireCreator` have no common resources that they receive on instantiation. This is possible due to the ability of Wyvern's module system to support a certain level of

```
1  // module definitions
2  module def darkTheme(l: Logger, uim: UIManager):
3                      {} Plugin[{l.Update, uim.PaintUI, uim.SetLookAndFeel}]
4
5  effect Run = {l.Update, uim.PaintUI, uim.SetUILookAndFeel}
6  ...
7
8  module def questionnaireCreator(l: Logger, ta: TextArea, fs: FileSystem):
9                  {} Plugin[{l.Update, fs.Write, fs.Append, ta.ReadTextArea}]
10
11 effect Run = {l.Update, fs.Write, fs.Append, ta.ReadTextArea}
12 ...
13
14 // module instantiations
15 val dt: Plugin[{l.Update, uim.PaintUI, uim.SetLookAndFeel}] =
16                                                          darkTheme(l, uim)
17 val qc: Plugin[{l.Update, fs.Write, fs.Append, ta.ReadTextArea}] =
18                                              questionnaireCreator(l, ta, fs)
```

Figure 4.9: Code snippets of the `darkTheme` and `questionnaireCreator` plugins. For simplicity, we substituted globally available UI-related effects with effects from the corresponding UI-related objects.

type abstraction, namely, that types do not include information about what resources are passed into modules on their instantiation.

Furthermore, since the resources that the plugins use differ, so do the effects that running the plugins' `run` method produces, which is handled by the effect-abstraction feature of our effect-system design. Specifically, since the `Run` effect in the `Plugin` type is abstract, the definitions of that effect in the two plugins can be and, in fact, are different (see lines 5 and 11 in the code snippets), accommodating the difference in the resource usage.

Finally, to account for the difference in the effects that running the two plugins causes in the code outside the modules, we use effect polymorphism (more details about which can be found in [36]). In particular, the `Plugin` type is parametric in that it is parameterized with the effects used to define the `Run` effect in the modules' return types (lines 3 and 9 in the code snippets) as well as at the place of instantiation (lines 15–18 in the code snippets).

The second example lies in the logging module. In the current version of the text-editor code, the `logger` module is implemented using the file system and stores the log file locally in a file. In the future, the text-editor application can be made distributed, so that files that the application operates on can be stored not locally but on some machine in the network. Similarly, in such a version of the text-editor application, the logging module could maintain a log file which is stored somewhere else on the network (e.g., as was suggested in Section 3.3.2). Due to the type abstraction in terms of what resources a module receives on instantiation and effect abstraction, this change to the application is easily accommodated in the current version of the text editor's code. As long as the new, distributed logger implements the `Logger` type, the modules that use `logger` are not affected by the substitution.

There is one more possible change that our design can accommodate. In the current version of the text editor's code, the `logger` module only appends to the log file, thus producing the `Append` effect on the `logFile` module, which is reflected in the definition of `logger`'s `Update` effect. Alternatively, `logger` could write to the log file aggregating the information that has been already logged, e.g., substituting "X action occurred. X action occurred." with "X action occurred 2 times." In such a case, `logger` would produce the `Write` effect on the `logFile` module, and the definition of `logger`'s `Update` effect would change accordingly. Due to effect abstraction, there would be no difference for the modules that use the `logger` module, which would still produce `logger`'s `Update` effect.

Thus, our case study demonstrated the usefulness of the type-abstraction, effect-abstraction, and effect-polymorphism features of our module-system and effect-system designs, and also provided more real-life examples for the information-hiding pattern that we suggested to use when developing a software application in Wyvern.

Enforcing That a Method May Produce No Effects. Our effect-system design allows a software developer to express the intent that a method may not produce any effects, which is then enforced by Wyvern's effect system. If the software application is fully effect-annotated and effects account for all important operations on system resources, the ability to specify and enforce that a method may cause no effects is equivalent to the ability to disallow the use of any resource inside that method. Inside of it, the method could still have an object capability to access a resource, but the method may not use that capability to perform any operations on the resource.

During the implementation of the text-editor application, we took advantage of this feature. When defining the `Plugin` type, which all plugins must implement, we needed to add a method that returns the plugin's name (line 3 in Figure 4.2), so that the plugin can be added to the text editor's user menu. All that such method needs to do is to return a `String` with the name, and thus the method must use no resources. To enforce this restriction, in the `Plugin` type, we annotated the method with `{}`, i.e., an empty effect set, which precluded any operations on resources inside the method.

Therefore, it proved useful to have a mechanism in our effect-system design that prohibits a method from having any effects, thus, potentially, disallowing it to operate on any resources inside of it.

Detecting the Authority-Attenuation Pattern. Section 3.5.3 describes how authority attenuation can be formalized using our effect-system design. In practice, as suggested in Section 3.3.4, since method effect annotations expose the information about how resources are used, a software developer is able to identify occurrences of authority attenuation by looking at modules' interfaces.

In the text-editor application, the `logger` module attenuates the `logFile` module. This can be seen in the diagram in Figure 4.6 from the effects of the two modules and the arrows that go in and out of the modules. Specifically, one can observe that, while `logFile` has three effects, namely, `Read`, `Write`, and `Append`, `logger` produces only one of them, namely, the `Append` effect. Also, there is only one arrow going into the `logFile` box, which is from the `logger` box, whereas there are several arrows that go into the `logger` box. This means that all the modules that modify the log file do it through the `logger`

	Effects per definition	Effects per annotation
Text editor	1.6	1.5
Plugins	3.3	0.9
Overall	1.8	1.3

Table 4.1: The average number (arithmetic mean) of effects per an effect set.

module, which allows for only a limited set of effects to be produced on `logFile`, thus attenuating it.

Thus, while analyzing the text-editor application, we were able to identify a real-life example of the authority-attenuation pattern and did that by examining only module interfaces and not their code. Considering the structured nature of the module interfaces, we believe that it is feasible to automate this discovery process.

Effect Aggregation. Table 4.1 shows the average number (arithmetic mean[6]) of effects in each effect set used in the implementation. This aspect speaks to the amount of boilerplate code that the effect-aggregation feature of our effect-system design eliminates.

Interestingly, the average number of effects in the effect-definition sets is much lower for the text editor than for the plugins, which signals that usually effects declared in the text editor are composed of fewer effects than those declared in plugins. There are at least two reasons for that. The main reason is that text editor's methods frequently use only one resource each and perform only one operation on it, whereas, in a plugin, the `run` method tends to use all the resources that the plugin has access to. Another, minor reason is that the `textEditor` module defines an effect, called `SaveFile`, whose definition consists of four effects, which is then used as a shorthand in defining two out of seven `textEditor`'s effects. The text editor's code base being relatively small, using the `SaveFile` effect as a shorthand may have a role in the lower average number of effects per definition.

In contrast to effect definitions, the difference between the average numbers of effects in effect-annotation sets in the text editor and the plugins is insignificant, and the numbers are low. For the text-editor application, the reason why the number of effects per effect annotation is low is that the same `SaveFile` is used to annotate five out of fifteen (i.e., one third of) `textEditor`'s methods. As expected, all five methods are related to saving a file to the disk. In addition, three more `textEditor`'s methods have empty effect annotations. For the plugins, the number of effects per effect annotation is low because there is only one method (the `run` method) that has an effect annotation with an effect (`Plugin`'s `Run` effect) in it, and the rest of the methods have empty effect annotations.

Overall, these observations imply that the effect-aggregation feature has its merit and indeed serves to reduce the amount of effect-related code.

Compiler Help with Method Effect Annotations. During the development of the text-editor application, the compiler was useful in ensuring that all the effects were captured by the

[6]We did not observe any strong outliers, and so the arithmetic mean was sufficient to summarize the data (e.g., as opposed to using the geometric mean).

```
1  // From uiEffects.wyv
2  module uiEffects: {}
3  effect ReadTextArea = {system.FFI}
4
5  // TextArea.wyt (in full)
6  import uiEffects
7  resource type TextArea
8    effect Read = {uiEffects.ReadTextArea}
9    def getText(): {Read} String
10
11 // From textEditor.wyv
12 import java:wyvern.stdlib.support.TextEditorHelper.nativeJTextArea
13 val textArea = nativeJTextArea.create(20, 60)
14
15 def getAttenuatedNativeJTextArea(): {} TextArea
16   new
17     effect Read = {uiEffects.ReadTextArea}
18     def getText(): {Read} String
19       textArea.getText()
20
21 val wc = wordCount(logger, getAttenuatedNativeJTextArea(),
22                                          getAttenuatedNativeJOptionPane())
```

Figure 4.10: Code snippets relevant to attenuating Java's `textArea` object.

method effect annotations. There were several instances when we forgot to include an effect in a method effect annotation, and the compiler gave an error pointing that out.

Since the compiler is able to provide such help with the effect-annotation inference, it may be possible to use this compiler feature to create a more advanced version of the compiler or a standalone tool that would be able to automatically infer and add some or all method effect annotations.

Limitations

During the implementation and analysis of the text-editor applications and its plugins, we observed the following two limitations of our current approach.

Non-Capability-Based Backend. As was mentioned in Section 4.2.1, we developed the text-editor application using Wyvern's Java backend. Java is not a capability-based programming language and does not support effects (at least not in the same sense as Wyvern does), and so, to minimize the security risks that come from using the Java backend, one must handle method calls into the backend and the overall Wyvern-to-backend communication carefully. In the text editor implementation, to secure the interlanguage communication, we took the following three measures.

The first measure results from the fact that we need a way to account for the effects that are caused by the Java code. To achieve that, we created a special effect, called

```
1 def registerPlugin(plugin: Plugin): {RegisterPlugin, plugin.Run} Unit
2   plugins.add(nativeActionCreator.createWithAction(plugin.getName(),
3               () => plugin.run()))
```

Figure 4.11: The `registerPlugin` method of the `textEditor` module.

`system.FFI`, which we use to annotate all methods that call into Java.[7] For example, consider the code snippets in Figure 4.10. The `getText` method (lines 19–20) calls into Java by calling the `getText` method (line 20) on the `textArea` object that represents the text-area object in Java (lines 12–13). To account for this call into the Java backend code, the `getText` method is said to have the `system.FFI` effect, which can be seen if we resolve the higher-level `Read` effect to lower-level effects (line 17 and then line 3).

The second measure is that we attenuate Java objects before passing them to plugins. Specifically, we wrapped Java objects into Wyvern objects making available only the methods that plugins need for their functionality and nothing more. Also, since we know what methods plugins call on the Java objects, instead of using the `system.FFI` effect described above, we introduced more descriptive effects. For example, Figure 4.10 presents this attenuation process for the Java object representing the text area of the text-editor application. First, we create the `TextArea` type (lines 6–9) that contains only the methods that plugins may use, namely, the `getText` method (line 9). We also introduce the `Read` effect (line 8) and use it to annotate the `getText` method, thus communicating the information that, if the `getText` method is called, the text area is being read. Then, we create a wrapper object for the original text-area object (lines 16–19) and ascribe it the `TextArea` type (line 15). All the plugins, e.g., `wordCount` (lines 21–22), are given the attenuated version of the original Java object. Thus, we made available only a limited set of the methods of the original Java objects and introduced more descriptive effects to better capture how the Java objects are used by the plugins.

The third measure is related to how plugin objects are used in the text-editor application. The plugins that are registered with the text editor's user menu must be forwarded onto the Java code. This is done in the `registerPlugin` method of the `textEditor` module, which is shown in Figure 4.11. For each plugin that is being registered, its `run` method is passed to Java as a lambda (line 3). Once the `run` method is passed to Java, there is no way to track the effects that the `run` method produces. To mitigate this issue, before the `run` method escapes into Java, we expose its effects by annotating with them the `registerPlugin` method (line 1). Notably, this is imprecise because we treat the effects as if they happened at the registration time, whereas in reality they happen later, when the callback is invoked. In the future, this could be improved by translating more of the Java backend code into Wyvern, including the code that handles the user-menu actions.

In general, although Wyvern does not provide inherently unsafe operations, such as pointers in C or `unsafe` in Rust, there are still risks associated with the non-capability-based backend, and method calls going to and coming from the backend must be imple-

[7] Currently, the annotations are added by hand, and the compiler does not check them, but we plan to automate adding the annotations and implement the compiler check in the near future.

	LoC	Effect declarations		Effect annotations		Effect parameters		Total	
Text editor	233	29	(12%)	57	(24%)	3	(1%)	86	(37%)
Plugins	110	3	(3%)	12	(11%)	6*	(5%)	21	(19%)
Total	343	32	(9%)	69	(20%)	9	(3%)	110	(32%)

Table 4.2: The effect-annotation overhead in the text-editor application and its plugins. For the plugins, the number of lines that contain effect parameters, marked with an asterisk, includes the lines where plugins are instantiated that are located in the text editor's code.

mented with utmost care. The main strength of Wyvern coming from its object model, there must be no way of getting around it.

Effect-Annotation Overhead. There are two ways to analyze the effect-annotation overhead. One way, which is higher-level, is in terms of the number of lines of code that are effected by adding the information about the code's effects. The other way, which is lower-level, is in terms of examining how the affected lines of code were changed.

Table 4.2 presents the higher-level picture about the effect-annotation overhead. Overall, the effect-annotation overhead comes from three sources: effect declarations, effect annotations on methods, and effect parameters. The important distinction among these types of effect-annotation overhead is that effect declarations require adding new lines of code to the implementation, whereas effect annotations and effect parameters changes the lines of code that would exist in an unannotated version of code.

Incorporating effects into the text editor's code base led to a 9%-increase of its size and affected 23% of (the enlarged version of) it, and so, overall, 32% of code was affected by the inclusion of the effect information. The percent is lower for the plugins than for the text-editor application itself (19% vs. 37%). The reason for this difference is that the text-editor application possesses and exercises more authority. The text editor accesses and operates on a larger number of resources than any one plugin does, and also the application defines the effects that the plugins may have. In contrast to the text editor, all three plugins define and use only one effect that involves using resources, the `Run` effect, (and one more empty effect) which, by our design of the text-editor application, is intended to support the desired authority restrictions. Also, none of the plugins introduce new effects that would be pertinent only to the plugin itself.

To perform the lower-level analysis of the effect annotations, we need to examine how each of the three sources of effect-annotation overhead affects the changed lines of code. For effect declarations and effect annotations, this aspect has been studied above as part of the discussion about effect aggregation, which was demonstrated to be useful for minimizing the code boilerplate. Here we discuss the remaining source of effect annotations, i.e., effect parameters.

The effect-parameterization feature of Wyvern's effect system was discussed in more detail elsewhere [36]. Part of the reason for introducing the feature into Wyvern is to overcome the issue of not being able to always guarantee that the objects to be used in effect annotations are in scope. In the text editor implementation, we solved this issue by

```
1  effect Run = {logger.Update, system.FFI, uiEffects.PaintUI,
2                uiEffects.SetUILookAndFeel, uiEffects.ReadTextArea,
3                uiEffects.ShowDialog, fs.Write, fs.Append}
4
5  def performNewAction(): {Run} Unit
6    val te: TextEditor[{logger.Update, system.FFI, uiEffects.PaintUI,
7                        uiEffects.SetUILookAndFeel, uiEffects.ReadTextArea,
8                        uiEffects.ShowDialog, fs.Write, fs.Append}] =
9                                                  createTextEditorInstance()
10   te.run()
```

Figure 4.12: The `performNewAction` method and the `Run` effect definitions of the `textEditor` module.

combining effect parameterization with globally available effects. Notably, this issue led us to use globally available UI-related effects as described in Section 4.2.2 and also helped us discover this new use case for the globally available effects, which adds to the use case described in Section 3.3.3.

As an example for when it would not be possible to guarantee that the objects to be used in effect annotations are in scope, consider the `performNewAction` method of the `textEditor` module, the relevant code snippets for which are shown in Figure 4.12. This method is called when a new text-editor window is opened. The `performNewAction` method creates a new `textEditor` instance (lines 2–5) and then runs it (line 6). When the new `textEditor` instance is run, we need to account for all effects that running it produces, including the effects that running its plugins produces. However, since plugins are instantiated inside the `textEditor` instance, in the `performNewAction` method, where `textEditor` is instantiated, there are no plugin objects available to use in effect annotations. To be able to provide the necessary effect information, the `TextEditor` type is parametric, and its parameters include the globally available UI-effects, as can be seen in lines 6–8 in the code snippets.

Although the issue is resolved from the technical standpoint, due to the verbosity of the effect parameters of the `TextEditor` type, the solution leads to high effect-annotation overhead. Specifically, the line of code on which a new instance of the text-editor application is created, which is split between lines 6–9 in Figure 4.12, is the second longest line in the implementation and has 192 characters.

In fact, the three longest lines in the text editor's code, which are 409, 192, and 186 characters long, respectively, are the result of effect parametricity. The longest line is the module header of the `textEditor` module, presented in Figure 4.13. It contains two effect-parametric types: the return type of the module and also the return type of the lambda that is passed into the module functor as an argument. Both effect sets used as effect parameters in the effect-parametric return types have seven effects each. The second longest line is described above, and the third longest line is the method signature of the `createTextEditorInstance` method. A call to the method can be seen on line 9 of the code snippet in Figure 4.12. The method has an effect-parametric return type matching the

```
module def textEditor(java: Java,
                      fs: fileSystem.FileSystem,
                      logger: Logger,
                      createTextEditorInstance:
                             Unit -> TextEditor[{logger.Update,
                                                 uiEffects.PaintUI,
                                                 uiEffects.SetUILookAndFeel,
                                                 uiEffects.ReadTextArea,
                                                 uiEffects.ShowDialog,
                                                 fs.Write,
                                                 fs.Append}]
                      ): {system.FFI} TextEditor[{logger.Update,
                                                  uiEffects.PaintUI,
                                                  uiEffects.SetUILookAndFeel,
                                                  uiEffects.ReadTextArea,
                                                  uiEffects.ShowDialog,
                                                  fs.Write,
                                                  fs.Append}]
```

Figure 4.13: The module header of the `textEditor` module, which is the longest line in the text editor's implementation and has 409 characters.

type of the `te` variable in the code snippet. All three longest lines are in the code of the text editor itself.

While effect-parametricity may cause effects to be verbose, there is a security benefit to using effect parameters. Namely, specifying effects in types allows for a greater exposure of the modules' authority in their interfaces. For example, consider the longest line of code in the text editor application, shown in Figure 4.13. Due to the effect parameters, we can account for all the effects that creating a `textEditor` module instance causes, which gives us a fuller picture of the `textEditor`'s authority.

Also, we see two possible mitigations to the issue of effect verbosity. One mitigation is allowing splitting Wyvern statements into multiple lines, e.g., by following Python's way of using implied line continuations inside parentheses, brackets, and braces. Another mitigation is to use more coarse-grained effects. For example, in the text-editor application, the UI-related effects could be defined to be a single effect or broken down into fewer distinct effects. However, this mitigation involves a tradeoff between code readability and maintainability, and security, which would be a design choice for the software developer writing the application.

All in all, adding information about code's effects affected about a third of the text editor's code. The text-editor application bears most of the effect-annotation burden, whereas the burden on the plugins is light. Thus, we expect the ratio of affected lines to go down as more plugins are added, and also effect annotations not to be a deterrent for plugin developers. In addition, while the effect verbosity may be a concern, there are mitigations for it.

Chapter 5

Conclusion and Future Work

Writing code that adheres to the principle of least privilege is an important yet challenging step towards creating more secure software applications. We created two programming-language mechanisms which simplify this task. Namely, we developed a capability-based module system that allows restricting and controlling access to system resources (Chapter 2), and also, we developed an effect system that allows manifesting and controlling how system resources are used in a program (Chapter 3). Using a case study of a text-editor application, we demonstrated that both of these programming-language features together aid a software developer in expressing security- and privacy-related constrains when writing a program and aid a security analyst in verifying a program's security during a security analysis (Chapter 4). Concluding this dissertation, we highlight the contributions that we made and suggest some paths for future work.

5.1 Contributions

The contributions of this work are:

- The design of a module system that supports first-class modules (cf. Newspeak, Scala, and Grace) and is capability safe [45, 48]. Our approach forbids ambient authority [73], instead requiring each module to take the resources it needs as parameters (similar to Newspeak, but in contrast to Scala and Grace). For practical purposes, our module system supports module-local state and does not restrict the imports of non-state-bearing modules (in contrast to Newspeak).
- A type system that distinguishes modules and objects that act as capabilities to access sensitive resources, from modules and objects that are purely functional computation or store immutable data. This design makes it easy for an architect to focus on the parts of an interface that are relevant to the authority of a module. Overall, the type system allows software developers to determine the authority of a module at compile time by examining only the interfaces of the module and the modules it imports, without having to look at the implementation of the involved modules.
- The formalization of capability control in the designed module system, in which we introduce a novel, non-transitive definition of capabilities. We also introduce a definition of capability safety and formally prove the designed system capability safe. Our result

contrasts with prior, transitive definitions of capability safety [17, 41].

- The design of an effect system that supports effect abstraction and provides a means for controlling and reasoning about module authority without having to sacrifice programming language expressiveness. In our effect system, effects are defined in terms of object capabilities allowing for a tighter connection between the operations performed on a resource (i.e., authority over a resource) and the resource itself. In addition, using our effect system, in most cases, software developers and security analysts are able to determine authority of a module by looking only at its interface, not needing to examine its code.
- The application of our module-system and effect-system designs to a number of common software-development patterns and forms of security reasoning. In each case, we illustrated the ability of our designs to express and enforce various security and software-architecture constraints about the software system under development.
- The formalization of our effect system, which includes the definition of authority safety and the proof of our programming language being authority safe. We also, for the first time, provided a formal definition of a common object-capability pattern of authority attenuation [48], allowing its inclusion in formal reasoning about a software application's security.
- The implementation of the designed module system and effect system in Wyvern, a statically typed, capability-safe, object-oriented programming language [52], demonstrating the feasibility and practicality of the proposed approach.
- A case study based on the implementation in Wyvern of an extensible text-editor application and plugins for it. The application is able to perform the basic text-editing functionality and serves as an example of how our module-system and effect-system designs serve to express and verify security constraints in a Wyvern application.

5.2 Future Work

There are several ways in which the work presented in this dissertation could be extended:

- Controlling system-resource use in a large software system is tedious, time consuming, and error prone. To help a software developer or a security analyst in this task, we could develop a tool that would extract relevant information from module interfaces, perform an analysis of capabilities and effect annotations in them, and automatically display that information in a convenient format. For example, such a tool could automatically identify instances of the authority-attenuation patterns, similar to those described in Sections 3.3.4 and 3.5.3, and manually identified in Section 4.2.3.
- As described in Section 4.2.3, currently, Wyvern's compiler is able to give errors when a method effect annotation is missing. We could further extend the compiler to perform a full inference of effect annotations, which would help software developers in annotating a large software system or gradually annotating previously unannotated code.
- To enrich the expressiveness of Wyvern's effects, building on the subtyping rules described in Section 3.4.7, we could develop a more elaborate effect subtyping. For example, we could create a subtyping mechanism using which it would be possible to express that one

effect is equivalent to another without having to provide a definition for the former one and allow substituting the latter effect for the former one.

- As was discussed in Section 4.2.3, there is a challenge in maintaining the desired level of security in a Wyvern application while relying on non-capability-based backend. To protect Wyvern applications more in-depth, we could reimplement the key pieces of the backend in Wyvern or reinforce the layer between Wyvern and the current backend languages:
  - For Java, we could compile the Wyvern code that interoperates with the backend Java code into bytecode with appropriately set Java-Security-Manager restrictions [56], and
  - For JavaScript, we could compile the Wyvern code that interoperates with the backend JavaScript code into Secure EcmaScript [2].
- To account for changes during the program execution more precisely, we could add a run-time notion of effects to Wyvern's dynamic semantics (the version presented in Section 3.4.6), e.g., using the most simplistic one [69].
- In this dissertation, we analyzed the application of our module-system and effect-system designs only to the security domain. However, historically, capabilities, module systems, and effect systems were used in various other domains and for various software-engineering concerns. It would be interesting and valuable to explore how our programming-language designs perform in other domains, e.g., concurrency and memory management, and accommodate other software-engineering concerns, e.g., modularity, integrity, reusability, and code maintainability.

Appendix A

Capability-Safe Module System

A.1 Type Soundness

A.1.1 Preservation

Lemma 1 (Preservation of types under substitution)**.** *If* $\Gamma,\ z : \tau' \mid \Sigma \vdash^{e''} e : \tau$ *and* $\Gamma \mid \Sigma \vdash^{e''} e' : \tau'$*, then* $\Gamma \mid \Sigma \vdash^{e''} [e'/z]e : \tau$*. Furthermore, if* $\Gamma,\ z : \tau' \mid \Sigma \vdash_s^{x'} d : \sigma$ *and* $\Gamma \mid \Sigma \vdash^{x'} e' : \tau'$*, then* $\Gamma \mid \Sigma \vdash_s^{x'} [e'/z]d : \sigma$.

Proof. The proof is by simultaneous induction on a derivation of $\Gamma,\ z : \tau' \mid \Sigma \vdash^{e''} e : \tau$ and $\Gamma,\ z : \tau' \mid \Sigma \vdash_s^{x'} d : \sigma$. 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 $[e'/z]x = e'$. The required result is then $\Gamma \mid \Sigma \vdash^{e''} e' : \tau'$, which is among the assumptions of the lemma. Otherwise, $[e'/z]x = x$, and the desired result is immediate.

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

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

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

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

Case T-BIND*:* $e = \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 : \tau_2$, and $[e'/z](\texttt{bind}\ x = e_1\ \texttt{in}\ e_2) = \texttt{bind}\ x = [e'/z]e_1\ \texttt{in}\ e_2$. By inversion on T-BIND, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash^{e''} e_1 : \tau_1$, and by the IH, $\Gamma \mid \Sigma \vdash^{e''} [e'/z]e_1 : \tau_1$. Then, by T-BIND, $\Gamma \mid \Sigma \vdash^{e''} \texttt{bind}\ x = [e'/z]e_1\ \texttt{in}\ e_2 : \tau_2$, i.e., $\Gamma \mid \Sigma \vdash^{e''} [e'/z](\texttt{bind}\ x = e_1\ \texttt{in}\ e_2) : \tau_2$.

Case T-LOC*:* $e = l$, $[e'/z]l = l$, and the desired result is immediate.

Case T-STACKFRAME*:* $e = l_1.m(l_2) \rhd e_1$, and by inversion on T-STACKFRAME, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash^{e''} l_1 : \{\overline{\sigma}\}_s$; $\texttt{def}\ m(x : \tau_2) : \tau_1 \in \overline{\sigma}$; $\Gamma,\ z : \tau' \mid \Sigma \vdash^{e''} l_2 : \tau_2$; and $\Gamma,\ l_2 : \tau_2,\ z : \tau' \mid \Sigma \vdash^{l_1} e_1 : \tau_1$. Locations are not affected by the substitution, and by the induction hypothesis, $\Gamma,\ l_2 : \tau_2 \mid \Sigma \vdash^{e''} [e'/z]e_1 : \tau_1$. Then, by T-STACKFRAME, $\Gamma \mid \Sigma \vdash^{e''} l_1.m(l_2) \rhd [e'/z]e_1 : \tau_1$, i.e., $\Gamma \mid \Sigma \vdash^{e''} [e'/z](l_1.m(l_2) \rhd e_1) : \tau_1$.

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

Case DT-DECLS*:* By inversion on T-DECLS, we get $\forall j,\ d_j \in \overline{d},\ \sigma_j \in \overline{\sigma}, \Gamma, z : \tau' \mid \Sigma \vdash_s^{x'} d_j : \sigma_j$. By the IH, $\forall j,\ d_j \in \overline{d},\ \sigma_j \in \overline{\sigma},\ \Gamma \mid \Sigma \vdash_s^{x'} [e'/z]d_j : \sigma_j$. Then, by T-DECLS, $\Gamma \mid \Sigma \vdash_s^{x'} [e'/z]\overline{d} : \overline{\sigma}$.

Case DT-DEFPURE*:* $d = \texttt{def}\ m(y : \tau_1) : \tau_2 = e$. There are two subcases depending on whether z is in $\Gamma_{\texttt{pure}}$ or not.

Subcase $z \in \Gamma_{\texttt{pure}}$*:* By inversion on DT-DEFPURE, we get $\Gamma_{\texttt{resource}} = \{x : \{\overline{\sigma}\}_{\texttt{resource}} \mid x : \{\overline{\sigma}\}_{\texttt{resource}} \in \Gamma\}$; $\Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}}$; and $\Gamma_{\texttt{pure}},\ y : \tau_1 \mid \Sigma \vdash^{x'} e : \tau_2$, and the desired result is immediate.

Subcase $z \notin \Gamma_{\texttt{pure}}$*:* By inversion on DT-DEFPURE, we get $\Gamma_{\texttt{resource}} = \{x : \{\overline{\sigma}\}_{\texttt{resource}} \mid x : \{\overline{\sigma}\}_{\texttt{resource}} \in \Gamma\}$; $\Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}}$; and $\Gamma_{\texttt{pure}},\ y : \tau_1,\ z : \tau' \mid \Sigma \vdash^{x'} e : \tau_2$. By the IH, $\Gamma_{\texttt{pure}},\ y : \tau_1 \mid \Sigma \vdash^{x'} [e'/z]e : \tau_2$. Then, by DT-DEFPURE, $\Gamma \mid \Sigma \vdash_{\texttt{pure}}^{x'} \texttt{def}\ m(y : \tau_1) : \tau_2 = [e'/z]e\ :\ \texttt{def}\ m(y : \tau_1) : \tau_2$, i.e., $\Gamma \mid \Sigma \vdash_{\texttt{pure}}^{x'} [e'/z](\texttt{def}\ m(y : \tau_1) : \tau_2 = e)\ :\ \texttt{def}\ m(y : \tau_1) : \tau_2$.

Thus, in both cases, the type of d is preserved under substitution.

Case DT-DEFRESOURCE*:* $d = \texttt{def}\ m(x : \tau_1) : \tau_2 = e$, and by inversion on DT-DEFRESOURCE, we get $\Gamma, x : \tau_1,\ z : \tau' \mid \Sigma \vdash^{x'} e : \tau_2$. By the induction hypothesis, $\Gamma, x : \tau_1 \mid \Sigma \vdash^{x'} [e'/z]e : \tau_2$. Then, by DT-DEFRESOURCE, $\Gamma \mid \Sigma \vdash_{\texttt{resource}}^{x'} \texttt{def}\ m(x : \tau_1) : \tau_2 = [e'/z]e\ :\ \texttt{def}\ m(x : \tau_1) : \tau_2$, i.e., $\Gamma \mid \Sigma \vdash_{\texttt{resource}}^{x'} [e'/z](\texttt{def}\ m(x : \tau_1) : \tau_2 = e)\ :\ \texttt{def}\ m(x : \tau_1) : \tau_2$.

Case DT-VARX*:* $d = \texttt{var}\ f : \tau = x$, and by inversion on DT-VARX, we get

$\Gamma,\ z : \tau' \mid \Sigma \vdash^{x'} x : \tau$. There are two subcases to consider, depending on whether x is z or another variable. If $x = z$, then $\Gamma,\ z : \tau' \mid \Sigma \vdash^{x'} [e'/z]x : \tau$ yields $\Gamma,\ z : \tau' \mid \Sigma \vdash^{x'} e' : \tau$ and $\tau = \tau'$. Thus, $\Gamma \mid \Sigma \vdash^{x'}_{\texttt{resource}} \texttt{var}\ f : \tau = e' : \texttt{var}\ f : \tau$ as required. If $x \neq z$, then $\Gamma,\ z : \tau' \mid \Sigma \vdash^{x'} [e'/z]x : \tau$ yields $\Gamma,\ z : \tau' \mid \Sigma \vdash^{x'} x : \tau$, and the desired result is immediate.

Case DT-VARL*:* $d = \texttt{var}\ f : \tau = l$, i.e., the field is resolved to a location l. This is not affected by the substitution, and the desired result is immediate.

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

Theorem 1 (Preservation)**.** *If* $\Gamma \mid \Sigma \vdash^{e''} e : \tau$*,* $\mu : \Sigma$*, and* $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$*, then* $\exists \Sigma' \supseteq \Sigma$*,* $\mu' : \Sigma'$*, and* $\Gamma \mid \Sigma' \vdash^{e''} e' : \tau$*.*

Proof. The proof is by induction on a derivation of $\Gamma \mid \Sigma \vdash^{e''} e : \tau$. 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*:* $e = \texttt{new}_s(x \Rightarrow \overline{d})$, and by inversion on T-NEW, we get $\Gamma,\ x : \{\overline{\sigma}\}_s \mid \Sigma \vdash^x_s \overline{d} : \overline{\sigma}$. The store changes from μ to $\mu' = \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s$, 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. 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 $\mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s : \Sigma,\ l : \{\overline{\sigma}\}_s$, which implies that $\Sigma' = \Sigma,\ l : \{\overline{\sigma}\}_s$. Finally, by T-LOC, $\Gamma \mid \Sigma \vdash l : \{\overline{\sigma}\}_s$. 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 $\Gamma \mid \Sigma \vdash^{e''} l_1 : \{\overline{\sigma}\}_s$; $\texttt{def}\ m(x : \tau_2) : \tau_1 \in \overline{\sigma}$; and $\Gamma \mid \Sigma \vdash^{e''} l_2 : \tau_2$. The store μ does not change, and since T-STORE has been applied throughout, the store is well typed, and thus, $\Gamma \mid \Sigma \vdash^{e''}_s \texttt{def}\ m(l_2 : \tau_2) : \tau_1 = e : \texttt{def}\ m(x : \tau_2) : \tau_1$. Then, by inversion on both DT-DEFPURE and DT-DEFRESOURCE, we know that $\Gamma,\ l_2 : \tau_2 \mid \Sigma \vdash^{e''} e : \tau_1$, and by T-STACKFRAME, we have $\Gamma,\ l_2 : \tau_2 \mid \Sigma \vdash^{e''} l_1.m(l_2) \rhd e : \tau_1$. Finally, by the preservation under subsumption lemma, substituting locations for variables in e preserve 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 $\Gamma \mid \Sigma \vdash^{l} l : \{\overline{\sigma}\}_s$ 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, $\Gamma \mid \Sigma \vdash_s^{l} \texttt{var}\ \ f : \tau = l_1\ :\ \texttt{var}\ \ f : \tau$. Then, by inversion on DT-VARL, we know that $\Gamma \mid \Sigma \vdash^{l} l_1 : \tau$, 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 $\Gamma \mid \Sigma \vdash^{l_1} l_1 : \{\overline{\sigma}\}_s$, $\texttt{var}\ \ f : \tau \in \overline{\sigma}$, and $\Gamma \mid \Sigma \vdash^{l_1} l_2 : \tau$. The store changes as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}_s / l_1 \mapsto \{x \Rightarrow \overline{d}\}_s]\mu$, 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, $\Gamma \mid \Sigma \vdash_s^{l_1} \texttt{var}\ \ f : \tau = l_2\ :\ \texttt{var}\ \ f : \tau$. Then, by inversion on DT-VARL, we know that $\Gamma \mid \Sigma \vdash^{l_1} l_2 : \tau$, and the right-hand side is well typed.

Case T-BIND*:* $e = \texttt{bind}\ x = e_1\ \texttt{in}\ e_2$, 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-BIND.

Subcase E-BIND*:* In this case, e_1 are values, namely locations l_1, and the result follows directly from the inversion on T-BIND and the preservation of types under substitution lemma.

Case T-STACKFRAME*:* $e = l.m(l_1) \rhd e_2$, and by the definition of the evaluation relation, there are two subcases:

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

Subcase E-STACKFRAME*:* In this case, e_2 is a value, i.e., a location l_2, and the result follows directly from the inversion on T-STACKFRAME.

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

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

A.1.2 Progress

Theorem 2 (Progress)**.** *If $\varnothing \mid \Sigma \vdash^{e''} e : \tau$ (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 $\Gamma \mid \Sigma \vdash^{e''} e : \tau$, 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*:* $e = \texttt{new}_s(x \Rightarrow \overline{d})$, and by E-NEW, e can make a step of evaluation if there is a location available that is not in the current store μ. 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 $\mu, l \mapsto \{x \Rightarrow \overline{d}\}_s$, 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 $\Gamma \mid \Sigma \vdash^{e''} e_1 : \{\overline{\sigma}\}_s$, either e_1 is a value or else it can make a step of evaluation, and, similarly, by the induction hypothesis applied to $\Gamma \mid \Sigma \vdash^{e''} e_2 : \tau_2$, 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 $\Gamma \mid \Sigma \vdash^{e''} l_1 : \{\overline{\sigma}\}_s$ and $\texttt{def}\ \ m(y : \tau_2) : \tau_1 \in \overline{\sigma}$. 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 $l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ with $\texttt{def}\ m(y : \tau_1) : \tau_2 = e \in \overline{d}$. 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 $\Gamma \mid \Sigma \vdash^{l} l : \{\overline{\sigma}\}_s$ 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 $l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ 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 $\Gamma \mid \Sigma \vdash^{l_1} l_1 : \{\overline{\sigma}\}_s$, $\texttt{var}\ \ f : \tau \in \overline{\sigma}$, and $\Gamma \mid \Sigma \vdash^{l_1} l_2 : \tau$. 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 $l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ with $\texttt{var}\ f : \tau = l \in \overline{d}$. A new well-typed store can be created as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}_s / l_1 \mapsto \{x \Rightarrow \overline{d}\}_s]\mu$, 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-BIND*:* $e = \texttt{bind}\ x = e_1\ \texttt{in}\ e_2$, 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 are values, i.e., locations l_1, the rule E-BIND applies, and e can take a step.

Case T-STACKFRAME*:* $e = l.m(l_1) \rhd e_2$, and by the induction hypothesis, either e_2 can make a step of evaluation or it is a value. Then, there are two subcases:

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

Subcase e_2 *is a value:* If e_2 is a value, i.e., a location l_2, the rule E-STACKFRAME applies, 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. □

A.2 Capability Safety

A.2.1 Capabilities-Related Properties

Property 3. *The runtime expression forms l and $l.m(l) \rhd e$ do not appear in the program source code.*

Proof. This property is enforced by the syntactic check of the source code of a program. □

Property 4. *Method-call stack frames ($l.m(l) \rhd e$) do not appear in method definitions and the bodies of the* `bind` *constructs.*

Proof. The proof is by induction over execution steps.
Base case: By Property 3, there are no method-call stack frames in the program source code.
Inductive case: The absence of method-call stack frames in the method definitions and the bodies of the `bind` constructs is maintained by all evaluation rules. Cases of E-METHOD and E-BIND involve substitution; however, substituted expression is a value (location), and thus, substitution preserves the property. □

Property 5. *Object fields are private to the objects they belong to and access to them can occur only inside methods of the objects to which they belong.*

Proof. The typing rules contain information about what object is (or will be, in case of an object creation) the receiver of the enclosing method. Then, from the T-FIELD and T-ASSIGN rules, it can be seen that, for a field access to occur, the receiver must be the object to which the field belongs. □

A.2.2 *subexps* Rules

$\boxed{subexps(E)}$

$$subexps([\,]) = \varnothing \quad \text{(SUBEXPS-EMPTY)}$$
$$subexps(E.m(e)) = \{e\} \cup subexps(E) \quad \text{(SUBEXPS-METHOD1)}$$
$$subexps(l.m(E)) = \{l\} \cup subexps(E) \quad \text{(SUBEXPS-METHOD2)}$$
$$subexps(E.f) = subexps(E) \quad \text{(SUBEXPS-FIELD)}$$
$$subexps(E.f = e) = \{e\} \cup subexps(E) \quad \text{(SUBEXPS-ASSIGN1)}$$
$$subexps(l.f = E) = \{l\} \cup subexps(E) \quad \text{(SUBEXPS-ASSIGN2)}$$
$$subexps(\texttt{bind}\ x = E\ \texttt{in}\ e) = \{e\} \cup subexps(E) \quad \text{(SUBEXPS-BIND)}$$
$$subexps(l.m(l') \rhd E) = \{l,\ l'\} \cup subexps(E) \quad \text{(SUBEXPS-STACKFRAME)}$$

A.2.3 Lemmas

Lemma 2. *If $l.m(l') \rhd E' \notin E$, then*

$$pointsto(E[e], \mu) = pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e', \mu).$$

Proof. The proof is by induction on E.
Case $E = [\,]$: $E[e] = e$

$$\begin{aligned} pointsto(E[e], \mu) &= pointsto(e, \mu) \\ &= pointsto(e, \mu) \cup \bigcup_{e' \in subexps([\,])} pointsto(e', \mu) \quad \text{(SUBEXPS-EMPTY)} \\ &= pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e', \mu) \end{aligned}$$

Case $E = E'.m(e'')$: $E[e] = E'[e].m(e'')$

$$\begin{aligned} subexps(E) &= subexps(E'.m(e'')) = \{e''\} \cup subexps(E') \quad \text{(SUBEXPS-METHOD1) [1]} \\ pointsto(E[e], \mu) &= pointsto(E'[e].m(e''), \mu) \\ &= pointsto(E'[e], \mu) \cup pointsto(e'', \mu) \quad \text{(POINTSTO-METHOD)} \\ &= pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E')} pointsto(e', \mu) \cup pointsto(e'', \mu) \quad (by\ IH) \\ &= pointsto(e, \mu) \cup \bigcup_{e' \in \{e''\} \cup subexps(E')} pointsto(e', \mu) \\ &= pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e', \mu) \quad (by\ [1]) \end{aligned}$$

Case $E = l.m(E')$: $E[e] = l.m(E'[e])$

$$
\begin{aligned}
subexps(E) &= subexps(l.m(E')) = \{l\} \cup subexps(E') && \text{(SUBEXPS-METHOD2) [2]}\\
pointsto(E[e],\mu) &= pointsto(l.m(E'[e]),\mu)\\
&= pointsto(l,\mu) \cup pointsto(E'[e],\mu) && \text{(POINTSTO-METHOD)}\\
&= pointsto(l,\mu) \cup pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E')} pointsto(e',\mu) && \textit{(by IH)}\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in \{l\} \cup subexps(E')} pointsto(e',\mu)\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e',\mu) && \textit{(by [2])}
\end{aligned}
$$

Case $E = E'.f$: $E[e] = E'[e].f$

$$
\begin{aligned}
subexps(E) &= subexps(E'.f) = subexps(E') && \text{(SUBEXPS-FIELD) [3]}\\
pointsto(E[e],\mu) &= pointsto(E'[e].f,\mu) = pointsto(E'[e],\mu) && \text{(POINTSTO-FIELD)}\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E')} pointsto(e',\mu) && \textit{(by IH)}\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e',\mu) && \textit{(by [3])}
\end{aligned}
$$

Case $E = (E'.f = e'')$: $E[e] = (E'[e].f = e'')$

$$
\begin{aligned}
subexps(E) &= subexps(E'.f = e'') = \{e''\} \cup subexps(E') && \text{(SUBEXPS-ASSIGN1) [4]}\\
pointsto(E[e],\mu) &= pointsto(E'[e].f = e'',\mu)\\
&= pointsto(E'[e],\mu) \cup pointsto(e'',\mu) && \text{(POINTSTO-ASSIGN)}\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E')} pointsto(e',\mu) \cup pointsto(e'',\mu) && \textit{(by IH)}\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in \{e''\} \cup subexps(E')} pointsto(e',\mu)\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e',\mu) && \textit{(by [4])}
\end{aligned}
$$

Case $E = (l.f = E')$: $E[e] = (l.f = E'[e])$

$$
\begin{aligned}
subexps(E) &= subexps(l.f = E') = \{l\} \cup subexps(E') && \text{(SUBEXPS-ASSIGN2) [5]}\\
pointsto(E[e],\mu) &= pointsto(l.f = E'[e],\mu)\\
&= pointsto(l,\mu) \cup pointsto(E'[e],\mu) && \text{(POINTSTO-ASSIGN)}\\
&= pointsto(l,\mu) \cup pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E')} pointsto(e',\mu) && \textit{(by IH)}\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in \{l\} \cup subexps(E')} pointsto(e',\mu)\\
&= pointsto(e,\mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e',\mu) && \textit{(by [5])}
\end{aligned}
$$

Case $E = (\texttt{bind}\ x = E'\ \texttt{in}\ e'')$: $E[e] = (\texttt{bind}\ x = E'[e]\ \texttt{in}\ e'')$

$$\begin{aligned}
subexps(E) &= subexps(\texttt{bind}\ x = E'\ \texttt{in}\ e'') \\
&= \{e''\} \cup subexps(E') && \text{(SUBEXPS-BIND) [6]} \\
pointsto(E[e], \mu) &= pointsto(\texttt{bind}\ x = E'[e]\ \texttt{in}\ e'', \mu) \\
&= pointsto(E'[e], \mu) \cup pointsto(e'', \mu) && \text{(POINTSTO-BIND)} \\
&= pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E')} pointsto(e', \mu) \cup pointsto(e'', \mu) \ \textit{(by IH)} \\
&= pointsto(e, \mu) \cup \bigcup_{e' \in \{e''\} \cup subexps(E')} pointsto(e', \mu) \\
&= pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e', \mu) && \textit{(by [6])}
\end{aligned}$$

Case $E = l.m(l') \rhd E'$: This case cannot happen as it contradicts the precondition that $l.m(l') \rhd E' \notin E$.

Thus, for all E, if $l.m(l') \rhd E' \notin E$, then

$$pointsto(E[e], \mu) = pointsto(e, \mu) \cup \bigcup_{e' \in subexps(E)} pointsto(e', \mu).$$

□

Lemma 3. *If*

1. *for* $1 \le i \le k$, $l.m(l') \rhd E \notin E_i$ [no method-call stack frames in E_i]
2. *for* $1 \le i \le k$, $l_i \mapsto \{x \Rightarrow \overline{d_i}\}_{\texttt{pure}} \in \mu$ [callers in all method-call stack frames are pure]

then $pointsto(E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$$= \bigcup_{i=1}^{k} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup pointsto(e, \mu).$$

Proof. The proof is by induction on the number of method-call stack frames preceding e on the stack.

Base case: $k = 1$

$pointsto(E_1[l_1.m_1(l'_1) \rhd e], \mu)$

$$= \bigcup_{e' \in subexps(E_1)} pointsto(e', \mu) \cup pointsto(e, \mu) \qquad (Lemma\ 2,\ \text{POINTSTO-CALL-PURE})$$

$$= \bigcup_{i=1}^{1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup pointsto(e, \mu)$$

Inductive case: $k > 1$

$pointsto(E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$$= \bigcup_{e' \in subexps(E_k)} pointsto(e', \mu) \qquad (Lemma\ 2,\ \text{POINTSTO-CALL-PURE})$$

$$\cup\ pointsto(E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$

$$= \bigcup_{e' \in subexps(E_k)} pointsto(e', \mu) \cup \bigcup_{e' \in subexps(E_{k-1})} pointsto(e', \mu)$$

$$\cup\ pointsto(E_{k-2}[l_{k-2}.m_{k-2}(l'_{k-2}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$

$$(Lemma\ 2,\ \text{POINTSTO-CALL-PURE})$$

$$= \bigcup_{i=1}^{k} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup pointsto(e, \mu)$$

$$((Lemma\ 2,\ \text{POINTSTO-CALL-PURE}) \times (k-2))$$

□

Lemma 4. *If*

1. *for* $1 \leq i \leq k$, $l.m(l') \rhd E \notin E_i$ [no method-call stack frames in E_i]
2. $\exists j$, *such that* $1 \leq j \leq k$, $l_j \mapsto \{x \Rightarrow \overline{d_j}\}_{\texttt{resource}} \in \mu$
   [there is at least one method-call stack frame that has a principal caller]

then $pointsto(E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$$= \bigcup_{i=p}^{k} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_p\},$$

where $1 \leq p \leq k$ *and* p *is the greatest index, such that* $l_p \mapsto \{x \Rightarrow \overline{d_p}\}_{\texttt{resource}} \in \mu$.
[l_p is the first (furthest from e) principal method caller on the stack]

Proof. The proof is by induction on the number of method-call stack frames preceding e on the stack.
Base case: $k = 1$, and since l_1 is the only method-call stack frame, $l_1 \mapsto \{x \Rightarrow \overline{d_1}\}_{\texttt{resource}} \in \mu$ and $p = 1$.
$pointsto(E_1[l_1.m_1(l'_1) \rhd e], \mu)$

$$= \bigcup_{e' \in subexps(E_1)} pointsto(e', \mu) \cup \{l_1\} \qquad (Lemma\ 2,\ \text{POINTSTO-CALL-PRINCIPAL})$$

$$= \bigcup_{i=1}^{1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_1\}$$

Inductive case: $k > 1$
$pointsto(E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$$= \bigcup_{e' \in subexps(E_k)} pointsto(e', \mu) \qquad (Lemma\ 2,\ \text{POINTSTO-CALL-PURE})$$
$$\cup\ pointsto(E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$
$$= \bigcup_{e' \in subexps(E_k)} pointsto(e', \mu) \cup \bigcup_{e' \in subexps(E_{k-1})} pointsto(e', \mu)$$
$$\cup\ pointsto(E_{k-2}[l_{k-2}.m_{k-2}(l'_{k-2}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$
$$(Lemma\ 2,\ \text{POINTSTO-CALL-PURE})$$
$$= \bigcup_{i=p+1}^{k} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu)$$
$$\cup\ pointsto(E_p[l_p.m_p(l'_p) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$
$$((Lemma\ 2,\ \text{POINTSTO-CALL-PURE}) \times (k - p - 2))$$
$$= \bigcup_{i=p}^{k} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_p\}$$
$$(Lemma\ 2,\ \text{POINTSTO-CALL-PRINCIPAL})$$

□

Lemma 5. *If* $l.m(l') \rhd E' \notin E$, *then* $cap_{stack}(l, E[e], \mu) = cap_{stack}(l, e, \mu)$.

Proof. Depending on whether $l.m(l') \rhd E' \in e$ or not, there are two possibilities.
Case $l.m(l') \rhd E' \in e$: $e = E''[l.m(l') \rhd e']$, where $l.m(l') \rhd E' \notin E''$, and $E[e] = E'''[l.m(l') \rhd e']$, where $E''' = E[E'']$ and $l.m(l') \rhd E' \notin E'''$.

$$\begin{aligned} cap_{stack}(l, E[e], \mu) &= cap_{stack}(l, E'''[l.m(l') \rhd e'], \mu) \\ &= pointsto(e', \mu) \cup cap_{stack}(l, e', \mu) && (\text{CAP-STACK}) \\ cap_{stack}(l, e, \mu) &= cap_{stack}(l, E''[l.m(l') \rhd e'], \mu) \\ &= pointsto(e', \mu) \cup cap_{stack}(l, e', \mu) && (\text{CAP-STACK}) \end{aligned}$$

Case $l.m(l') \rhd E' \notin e$: $l.m(l') \rhd E' \notin E[e]$.

$$\begin{aligned} cap_{stack}(l, E[e], \mu) &= \varnothing && (\text{CAP-STACK-NOCALL}) \\ cap_{stack}(l, e, \mu) &= \varnothing && (\text{CAP-STACK-NOCALL}) \end{aligned}$$

Thus, $cap_{stack}(l, E[e], \mu) = cap_{stack}(l, e, \mu)$. □

Lemma 6. *If*

1. *for* $1 \leq i \leq k$, $l'.m(l'') \rhd E \notin E_i$ [no method-call stack frames in E_i]
2. $l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu$ [l is a principal]
3. $\forall i$, *such that* $l_i = l$, $i \in \{q_1, q_2, \ldots, q_{r_1}\}$, *where* $0 \leq r_1 \leq k$
   [the set of indices of all method-call stack frames where l is the caller; this set can be empty]
4. $\forall i \in \{q_1, q_2, \ldots, q_{r_1}\}$, *if* $\exists j$, *such that*

(a) $l_j \mapsto \{x \Rightarrow \overline{d_j}\}_{\texttt{resource}} \in \mu$ *and* [l_j is a principal]

(b) $\forall t$*, such that* $i > t > j$ *and* $l_t \mapsto \{x \Rightarrow \overline{d_t}\}_{\texttt{pure}} \in \mu$
[all receivers between l_i and l_j are pure]

$j \in \{p_1, p_2, \ldots, p_{r_2}\}$ *where* $0 \leq r_2 \leq r_1$
[the maximal set of indices of principal callers immediately after method-call stack frames where l is the caller; this set can be smaller than the one above only by one element; this set can also be empty; such principals can be l itself]

then

$$cap_{stack}(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$

$$= \begin{cases} \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\ldots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e', \mu) & \text{if } r_2 < r_1 \\ \cup \bigcup_{j\in\{p_1,p_2,\ldots,p_{r_2}\}} \{l_j\} \cup \bigcup_{i=1}^{q_{r_2+1}-1} \bigcup_{e'\in subexps(E_i)} pointsto(e', \mu) & \\ \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) & \\ & \\ \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\ldots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e', \mu) & \text{if } r_2 = r_1 \\ \cup \bigcup_{j\in\{p_1,p_2,\ldots,p_{r_2}\}} \{l_j\} \cup cap_{stack}(l, e, \mu) & \end{cases}$$

[If $r_2 < r_1$, then there are only pure callers after the last method-call stack frame where l is the caller. In other words, l was the last principal caller on the stack.
If $r_2 = r_1$, then the last method-call stack frame where l is the caller is followed by a method-call stack frame with a principal caller that is not l. If $r_2 = r_1 = 0$, then there are no method-call stack frames with principal callers on the stack.
Since the set in 4(b) can include indices of method-call stack frames where the caller is l, the difference between r_1 and r_2 is at most 1, i.e., $r_2 \leq r_1 \leq r_2 + 1$.]

Proof. The proof is by induction on the number of method-call stack frames preceding e on the stack.

Base case: $k = 1$. Depending on the values of r_1 and r_2, there are two possibilities.
Case $r_2 < r_1$*:* $r_1 = 1$, $r_2 = 0$, $l_1 = l$, $q_1 = 1$, and $\nexists p_1$.

$$cap_{stack}(l, E_1[l_1.m_1(l'_1) \rhd e], \mu) = pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) \qquad \text{(CAP-STACK)}$$

Case $r_2 = r_1$*:* $r_1 = r_2 = 0$, $l_1 \neq l$, and $\nexists q_1, p_1$.

$$cap_{stack}(l, E_1[l_1.m_1(l'_1) \rhd e], \mu) = cap_{stack}(l, e, \mu) \qquad (Lemma\ 5)$$

Inductive case: $k > 1$
$cap_{stack}(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$= cap_{stack}(l, l_{q_1}.m_{q_1}(l'_{q_1}) \rhd E_{q_1-1}[l_{q_1-1}.m_{q_1-1}(l'_{q_1-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$(Lemma\ 5)$

$= pointsto(E_{q_1-1}[l_{q_1-1}.m_{q_1-1}(l'_{q_1-1}) \rhd E_{q_1-2}[l_{q_1-2}.m_{q_1-2}(l'_{q_1-2}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$\cup\ cap_{stack}(l, E_{q_1-1}[l_{q_1-1}.m_{q_1-1}(l'_{q_1-1}) \rhd E_{q_1-2}[l_{q_1-2}.m_{q_1-2}(l'_{q_1-2}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

(CAP-STACK)

$= \bigcup_{i=p_1}^{q_1-1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_{p_1}\}$

$\cup\ cap_{stack}(l, E_{q_1-1}[l_{q_1-1}.m_{q_1-1}(l'_{q_1-1}) \rhd E_{q_1-2}[l_{q_1-2}.m_{q_1-2}(l'_{q_1-2}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$(Lemma\ 4)$

$= \bigcup_{i=p_1}^{q_1-1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_{p_1}\}$

$\cup\ cap_{stack}(l, l_{q_2}.m_{q_2}(l'_{q_2}) \rhd E_{q_2-1}[l_{q_2-1}.m_{q_2-1}(l'_{q_2-1}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$(Lemma\ 5)$

$= \bigcup_{i=p_1}^{q_1-1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_{p_1}\}$

$\cup\ pointsto(E_{q_2-1}[l_{q_2-1}.m_{q_2-1}(l'_{q_2-1}) \rhd E_{q_2-2}[l_{q_2-2}.m_{q_2-2}(l'_{q_2-2}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$\cup\ cap_{stack}(l, E_{q_2-1}[l_{q_2-1}.m_{q_2-1}(l'_{q_2-1}) \rhd E_{q_2-2}[l_{q_2-2}.m_{q_2-2}(l'_{q_2-2}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

(CAP-STACK)

$= \bigcup_{i=p_1}^{q_1-1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_{p_1}\} \cup \bigcup_{i=p_2}^{q_2-1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \{l_{p_2}\}$

$\cup\ cap_{stack}(l, E_{q_2-1}[l_{q_2-1}.m_{q_2-1}(l'_{q_2-1}) \rhd E_{q_2-2}[l_{q_2-2}.m_{q_2-2}(l'_{q_2-2}) \rhd \ldots$

$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$(Lemma\ 4)$

$\vdots$

$= \bigcup_{(q,p) \in \{(q_1,p_1),(q_2,p_2),\ldots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e' \in subexps(E_i)} pointsto(e', \mu) \cup \bigcup_{j \in \{p_1,p_2,\ldots,p_{r_2}\}} \{l_j\}$

$\cup\ cap_{stack}(l, E_{q_{r_2}-1}[l_{q_{r_2}-1}.m_{q_{r_2}-1}(l'_{q_{r_2}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

Depending on the values of r_1 and r_2, there are two possibilities.

Case $r_2 < r_1$*:* There is no other resource callers after $l_{q_{r_2+1}}$, i.e., $\forall l_0.m_0(l'_0) \rhd E''' \in E_{q_{r_2+1}-1}[l_{q_{r_2+1}-1}.m_{q_{r_2+1}-1}(l'_{q_{r_2+1}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots]$, $l_0 \mapsto \{x \Rightarrow \overline{d_0}\}_{\texttt{pure}} \in \mu$, which implies that there are also no method-call stack frames with l as the caller after $l_{q_{r_2+1}}$, i.e., $l_1.m'(l'') \rhd E'' \notin E_{q_{r_2+1}-1}[l_{q_{r_2+1}-1}.m_{q_{r_2+1}-1}(l'_{q_{r_2+1}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots]$. Then,

$$cap_{stack}(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots], \mu)$$

$$= \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup \bigcup_{j\in\{p_1,p_2,\dots,p_{r_2}\}} \{l_j\}$$
$$\cup\, cap_{stack}(l, l_{q_{r_2+1}}.m_{q_{r_2+1}}(l'_{q_{r_2+1}}) \rhd E_{q_{r_2+1}-1}[l_{q_{r_2+1}-1}.m_{q_{r_2+1}-1}(l'_{q_{r_2+1}-1}) \rhd \dots$$
$$\cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots], \mu)$$
$$(Lemma\ 5)$$

$$= \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup \bigcup_{j\in\{p_1,p_2,\dots,p_{r_2}\}} \{l_j\}$$
$$\cup\, pointsto(E_{q_{r_2+1}-1}[l_{q_{r_2+1}-1}.m_{q_{r_2+1}-1}(l'_{q_{r_2+1}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots], \mu)$$
$$\cup\, cap_{stack}(l, E_{q_{r_2+1}-1}[l_{q_{r_2+1}-1}.m_{q_{r_2+1}-1}(l'_{q_{r_2+1}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots], \mu)$$
$$(\text{CAP-STACK})$$

$$= \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup \bigcup_{j\in\{p_1,p_2,\dots,p_{r_2}\}} \{l_j\}$$
$$\cup \bigcup_{i=1}^{q_{r_2+1}-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup pointsto(e,\mu)$$
$$\cup\, cap_{stack}(l, E_{q_{r_2+1}-1}[l_{q_{r_2+1}-1}.m_{q_{r_2+1}-1}(l'_{q_{r_2+1}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots], \mu)$$
$$(Lemma\ 3)$$

$$= \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup \bigcup_{j\in\{p_1,p_2,\dots,p_{r_2}\}} \{l_j\}$$
$$\cup \bigcup_{i=1}^{q_{r_2+1}-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup pointsto(e,\mu) \cup cap_{stack}(l,e,\mu)$$
$$(Lemma\ 5)$$

Case $r_2 = r_1$*:* There are no method-call stack frames with l as the caller after $l_{q_{r_2+1}}$, i.e., $l.m'(l'') \rhd E'' \notin E_{q_{r_2}-1}[l_{q_{r_2}-1}.m_{q_{r_2}-1}(l'_{q_{r_2}-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots]$

$$cap_{stack}(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l_1.m_1(l'_1) \rhd e]\dots], \mu)$$

$$= \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\ldots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'\in subexps(E_i)} pointsto(e',\mu) \cup \bigcup_{j\in\{p_1,p_2,\ldots,p_{r_2}\}} \{l_j\}$$
$$\cup\ cap_{stack}(l,e,\mu)$$
$$(Lemma\ 5)$$

□

Lemma 7. *If* $\langle E[e_0] \mid \mu\rangle \longrightarrow \langle E[e_0'] \mid \mu'\rangle$*, then*

$$\bigcup_{e\in subexps(E)} pointsto(e,\mu') = \bigcup_{e\in subexps(E)} pointsto(e,\mu).$$

Proof. The proof is by induction on the $subexps(E)$ rules.

Case SUBEXPS-EMPTY*:* Since the $subexps(E)$ returns an empty set, the desired result is immediate.

Case SUBEXPS-METHOD1*:*
$\bigcup_{e\in subexps(E.m(e''))} pointsto(e,\mu) = pointsto(e'',\mu) \cup \bigcup_{e\in subexps(E)} pointsto(e,\mu)$, and similarly, $\bigcup_{e\in subexps(E.m(e''))} pointsto(e,\mu') = pointsto(e'',\mu') \cup \bigcup_{e\in subexps(E)} pointsto(e,\mu')$.

Since we are considering small-step semantics and e'' is evaluated only after E is fully evaluated, there were no changes to e'' at this evaluation steps, and $pointsto(e'',\mu') = pointsto(e'',\mu)$. By the induction hypothesis, $\bigcup_{e\in subexps(E)} pointsto(e,\mu') = \bigcup_{e\in subexps(E)} pointsto(e,\mu)$.

Thus, $\bigcup_{e\in subexps(E.m(e''))} pointsto(e,\mu') = \bigcup_{e\in subexps(E.m(e''))} pointsto(e,\mu)$.

Case SUBEXPS-METHOD2*:*
$\bigcup_{e\in subexps(l.m(E))} pointsto(e,\mu) = pointsto(l,\mu) \cup \bigcup_{e\in subexps(E)} pointsto(e,\mu)$, and similarly, $\bigcup_{e\in subexps(l.m(E))} pointsto(e,\mu') = pointsto(l,\mu') \cup \bigcup_{e\in subexps(E)} pointsto(e,\mu')$.

By POINTSTO-PRINCIPAL and POINTSTO-PURE, $pointsto(l,\mu') = pointsto(l,\mu)$. By the induction hypothesis, $\bigcup_{e\in subexps(E)} pointsto(e,\mu') = \bigcup_{e\in subexps(E)} pointsto(e,\mu)$.

Thus, $\bigcup_{e\in subexps(l.m(E))} pointsto(e,\mu') = \bigcup_{e\in subexps(l.m(E))} pointsto(e,\mu)$.

Case SUBEXPS-FIELD*:* $\bigcup_{e\in subexps(E.f)} pointsto(e,\mu) = \bigcup_{e\in subexps(E)} pointsto(e,\mu)$, and similarly, $\bigcup_{e\in subexps(E.f)} pointsto(e,\mu') = \bigcup_{e\in subexps(E)} pointsto(e,\mu')$. By the induction hypothesis, $\bigcup_{e\in subexps(E)} pointsto(e,\mu') = \bigcup_{e\in subexps(E)} pointsto(e,\mu)$, and thus, $\bigcup_{e\in subexps(E.f)} pointsto(e,\mu') = \bigcup_{e\in subexps(E.f)} pointsto(e,\mu)$.

Case SUBEXPS-ASSIGN1*:*
$\bigcup_{e\in subexps(E.f=e'')} pointsto(e,\mu) = pointsto(e'',\mu) \cup \bigcup_{e\in subexps(E)} pointsto(e,\mu)$, and similarly, $\bigcup_{e\in subexps(E.f=e'')} pointsto(e,\mu') = pointsto(e'',\mu') \cup \bigcup_{e\in subexps(E)} pointsto(e,\mu')$.

Since we are considering small-step semantics and e'' is evaluated only after E is fully evaluated, there were no changes to e'' at this evaluation steps, and $pointsto(e'',\mu') = pointsto(e'',\mu)$. By the induction hypothesis, $\bigcup_{e\in subexps(E)} pointsto(e,\mu') = \bigcup_{e\in subexps(E)} pointsto(e,\mu)$.

Thus, $\bigcup_{e \in subexps(E.f=e'')} pointsto(e, \mu') = \bigcup_{e \in subexps(E.f=e'')} pointsto(e, \mu)$.

Case SUBEXPS-BIND*:*
$\bigcup_{e \in subexps(\texttt{bind } x=E \texttt{ in } e'')} pointsto(e, \mu) = pointsto(e'', \mu) \cup \bigcup_{e \in subexps(E)} pointsto(e, \mu)$ and $\bigcup_{e \in subexps(\texttt{bind } x=E \texttt{ in } e'')} pointsto(e, \mu') = pointsto(e'', \mu') \cup \bigcup_{e \in subexps(E)} pointsto(e, \mu')$.

Since we are considering small-step semantics and e'' is evaluated only after E is fully evaluated, there were no changes to e'' at this evaluation steps, and $pointsto(e'', \mu') = pointsto(e'', \mu)$. By the induction hypothesis, $\bigcup_{e \in subexps(E)} pointsto(e, \mu') = \bigcup_{e \in subexps(E)} pointsto(e, \mu)$.

Thus, $\bigcup_{e \in subexps(\texttt{bind } x=E \texttt{ in } e'')} pointsto(e, \mu') = \bigcup_{e \in subexps(\texttt{bind } x=E \texttt{ in } e'')} pointsto(e, \mu)$.

Case SUBEXPS-ASSIGN2*:*
$\bigcup_{e \in subexps(l.f=E)} pointsto(e, \mu) = pointsto(l, \mu) \cup \bigcup_{e \in subexps(E)} pointsto(e, \mu)$, and similarly, $\bigcup_{e \in subexps(l.f=E)} pointsto(e, \mu') = pointsto(l, \mu') \cup \bigcup_{e \in subexps(E)} pointsto(e, \mu')$.

By POINTSTO-PRINCIPAL and POINTSTO-PURE, $pointsto(l, \mu') = pointsto(l, \mu)$. By the induction hypothesis, $\bigcup_{e \in subexps(E)} pointsto(e, \mu') = \bigcup_{e \in subexps(E)} pointsto(e, \mu)$.

Thus, $\bigcup_{e \in subexps(l.f=E)} pointsto(e, \mu') = \bigcup_{e \in subexps(l.f=E)} pointsto(e, \mu)$.

Case SUBEXPS-STACKFRAME*:*
$\bigcup_{e \in subexps(l.m(l') \rhd E)} pointsto(e, \mu) = pointsto(l, \mu) \cup pointsto(l', \mu) \cup \bigcup_{e \in subexps(E)} pointsto(e, \mu)$, and similarly, $\bigcup_{e \in subexps(l.m(l') \rhd E)} pointsto(e, \mu') = pointsto(l, \mu') \cup pointsto(l', \mu') \cup \bigcup_{e \in subexps(E)} pointsto(e, \mu')$.

By POINTSTO-PRINCIPAL and POINTSTO-PURE, $pointsto(l, \mu') = pointsto(l, \mu)$ and $pointsto(l', \mu') = pointsto(l', \mu)$. By the induction hypothesis, $\bigcup_{e \in subexps(E)} pointsto(e, \mu') = \bigcup_{e \in subexps(E)} pointsto(e, \mu)$.

Thus, $\bigcup_{e \in subexps(l.m(l') \rhd E)} pointsto(e, \mu') = \bigcup_{e \in subexps(l.m(l') \rhd E)} pointsto(e, \mu)$. □

Lemma 8. *If*

1. $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$ [e can make a step of evaluation]
2. *for* $1 \leq i \leq k$, $l'.m(l'') \rhd E \notin E_i$ [no method-call stack frames in E_i]
3. $l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu$ [l is a principal]
4. $\forall i$, *such that* $l_i = l$, $i \in \{q_1, q_2, \ldots, q_{r_1}\}$, *where* $0 \leq r_1 \leq k$
   [the set of indices of all method-call stack frames where l is the caller; this set can be empty]
5. $\forall i \in \{q_1, q_2, \ldots, q_{r_1}\}$, *if* $\exists j$, *such that*
   (a) $l_j \mapsto \{x \Rightarrow \overline{d_j}\}_{\texttt{resource}} \in \mu$ *and* [l_j is a principal]
   (b) $\forall t$, *such that* $i > t > j$ *and* $l_t \mapsto \{x \Rightarrow \overline{d_t}\}_{\texttt{pure}} \in \mu$
   [all callers between l_i and l_j are pure]

   $j \in \{p_1, p_2, \ldots, p_{r_2}\}$ *where* $0 \leq r_2 \leq r_1$
   [the maximal set of indices of principal callers immediately after method-call stack frames where l is the caller; this set can be smaller than the one above only by one element; this set can also be empty; such principals can be l itself]

then
$cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e'] \ldots], \mu')$

$$\setminus\ cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$$

$$= \begin{cases} cap_{store}(l, \mu') \cup pointsto(e', \mu') \cup cap_{stack}(l, e', \mu') & \textit{if } r_2 < r_1 \\ \setminus\ cap_{store}(l, \mu) \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) & \\ cap_{store}(l, \mu') \cup cap_{stack}(l, e', \mu') & \textit{if } r_2 = r_1 \\ \setminus\ cap_{store}(l, \mu) \cup cap_{stack}(l, e, \mu) & \end{cases}$$

[If $r_2 < r_1$, then there are only pure callers after the last method-call stack frame where l is the caller. In other words, l was the last principal caller on the stack.
If $r_2 = r_1$, then the last method-call stack frame where l is the caller is followed by a method-call stack frame with a principal caller that is not l. If $r_2 = r_1 = 0$, then there are no method-call stack frames with principal callers on the stack.
Since the set in 5(b) can include indices of method-call stack frames where the caller is l, the difference between r_1 and r_2 is at most 1, i.e., $r_2 \leq r_1 \leq r_2 + 1$.]

Proof. The proof is by induction on the number of method-call stack frames preceding e and e' on the stack.

Base case: $k = 1$. Depending on the values of r_1 and r_2, there are two possibilities.
Case $r_2 < r_1$*:* $r_1 = 1$, $r_2 = 0$, $l_1 = l$, $q_1 = 1$, and $\nexists p_1$.
$cap(l, E_1[l_1.m_1(l'_1) \rhd e], \mu)$

$$\begin{aligned} &= cap_{store}(l, \mu) \cup cap_{stack}(l, E_1[l_1.m_1(l'_1) \rhd e], \mu) && \text{(CAP-CONFIG)} \\ &= cap_{store}(l, \mu) \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) && \text{(CAP-STACK)} \end{aligned}$$

Similarly, $cap(l, E_1[l_1.m_1(l'_1) \rhd e'], \mu') = cap_{store}(l, \mu') \cup pointsto(e', \mu') \cup cap_{stack}(l, e', \mu')$.
Then, $cap(l, E_1[l_1.m_1(l'_1) \rhd e'], \mu')\ \setminus\ cap(l, E_1[l_1.m_1(l'_1) \rhd e], \mu)$

$$\begin{aligned} &= cap_{store}(l, \mu') \cup pointsto(e', \mu') \cup cap_{stack}(l, e', \mu') \\ &\setminus\ cap_{store}(l, \mu) \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) \end{aligned}$$

Case $r_2 = r_1$*:* $r_1 = r_2 = 0$, $l_1 \neq l$, and $\nexists q_1, p_1$.
$cap(l, E_1[l_1.m_1(l'_1) \rhd e], \mu)$

$$\begin{aligned} &= cap_{store}(l, \mu) \cup cap_{stack}(l, E_1[l_1.m_1(l'_1) \rhd e], \mu) && \text{(CAP-CONFIG)} \\ &= cap_{store}(l, \mu) \cup cap_{stack}(l, e, \mu) && (Lemma\ 5) \end{aligned}$$

Similarly, $cap(l, E_1[l_1.m_1(l'_1) \rhd e'], \mu') = cap_{store}(l, \mu') \cup cap_{stack}(l, e', \mu')$. Then, $cap(l, E_1[l_1.m_1(l'_1) \rhd e'], \mu')\ \setminus\ cap(l, E_1[l_1.m_1(l'_1) \rhd e], \mu)$

$$= cap_{store}(l, \mu') \cup cap_{stack}(l, e', \mu')\ \setminus\ cap_{store}(l, \mu) \cup cap_{stack}(l, e, \mu)$$

Inductive case: $k > 1$
$cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l.m_1(l'_1) \rhd e] \ldots], \mu)$

$= cap_{store}(l, \mu) \cup cap_{stack}(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \dots$
$\cdots \rhd E_1[l.m_1(l'_1) \rhd e] \dots], \mu)$
(CAP-CONFIG)

$$= \begin{cases} cap_{store}(l, \mu) & \text{if } r_2 < r_1 \\ \cup \bigcup_{(q,p) \in \{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'' \in subexps(E_i)} pointsto(e'', \mu) \\ \cup \bigcup_{j \in \{p_1,p_2,\dots,p_{r_2}\}} \{l_j\} \cup \bigcup_{i=1}^{q_{r_2+1}-1} \bigcup_{e'' \in subexps(E_i)} pointsto(e'', \mu) \\ \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) \\ \\ cap_{store}(l, \mu) & \text{if } r_2 = r_1 \\ \cup \bigcup_{(q,p) \in \{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'' \in subexps(E_i)} pointsto(e'', \mu) \\ \cup \bigcup_{j \in \{p_1,p_2,\dots,p_{r_2}\}} \{l_j\} \cup cap_{stack}(l, e, \mu) \end{cases}$$

$(Lemma\ 6)$

Similarly, $cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l.m_1(l'_1) \rhd e'] \dots], \mu')$

$= cap_{store}(l, \mu') \cup cap_{stack}(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \dots$
$\cdots \rhd E_1[l.m_1(l'_1) \rhd e'] \dots], \mu')$
(CAP-CONFIG)

$$= \begin{cases} cap_{store}(l, \mu') & \text{if } r_2 < r_1 \\ \cup \bigcup_{(q,p) \in \{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'' \in subexps(E_i)} pointsto(e'', \mu') \\ \cup \bigcup_{j \in \{p_1,p_2,\dots,p_{r_2}\}} \{l_j\} \\ \cup \bigcup_{i=1}^{q_{r_2+1}-1} \bigcup_{e'' \in subexps(E_i)} pointsto(e'', \mu') \cup pointsto(e', \mu') \\ \cup cap_{stack}(l, e', \mu') \\ \\ cap_{store}(l, \mu') & \text{if } r_2 = r_1 \\ \cup \bigcup_{(q,p) \in \{(q_1,p_1),(q_2,p_2),\dots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e'' \in subexps(E_i)} pointsto(e'', \mu') \\ \cup \bigcup_{j \in \{p_1,p_2,\dots,p_{r_2}\}} \{l_j\} \\ \cup cap_{stack}(l, e', \mu') \end{cases}$$

$(Lemma\ 6)$

$$= \begin{cases} cap_{store}(l,\mu') & \text{if } r_2 < r_1 \\ \cup \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\ldots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e''\in subexps(E_i)} pointsto(e'',\mu) \\ \cup \bigcup_{j\in\{p_1,p_2,\ldots,p_{r_2}\}} \{l_j\} \\ \cup \bigcup_{i=1}^{q_{r_2+1}-1} \bigcup_{e''\in subexps(E_i)} pointsto(e'',\mu) \cup pointsto(e',\mu') \\ \cup cap_{stack}(l,e',\mu') \\ \\ cap_{store}(l,\mu') & \text{if } r_2 = r_1 \\ \cup \bigcup_{(q,p)\in\{(q_1,p_1),(q_2,p_2),\ldots(q_{r_2},p_{r_2})\}} \bigcup_{i=p}^{q-1} \bigcup_{e''\in subexps(E_i)} pointsto(e'',\mu) \\ \cup \bigcup_{j\in\{p_1,p_2,\ldots,p_{r_2}\}} \{l_j\} \\ \cup cap_{stack}(l,e',\mu') \end{cases}$$

$(Lemma\ 7)$

Then, $cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l.m_1(l'_1) \rhd e'] \ldots], \mu')$
$\setminus\ cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_1[l.m_1(l'_1) \rhd e] \ldots], \mu)$

$$= \begin{cases} cap_{store}(l,\mu') \cup pointsto(e',\mu') \cup cap_{stack}(l,e',\mu') & \text{if } r_2 < r_1 \\ \setminus\ cap_{store}(l,\mu) \cup pointsto(e,\mu) \cup cap_{stack}(l,e,\mu) \\ \\ cap_{store}(l,\mu') \cup cap_{stack}(l,e',\mu') & \text{if } r_2 = r_1 \\ \setminus\ cap_{store}(l,\mu) \cup cap_{stack}(l,e,\mu) \end{cases}$$

□

Lemma 9. *If* $l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ *and* $l'.m'(l'') \rhd E \notin e$*, then*

$$pointsto([l/z]e,\mu) = \begin{cases} pointsto(l,\mu) \cup pointsto(e,\mu) & \textit{if } z \in e \\ pointsto(e,\mu) & \textit{if } z \notin e \end{cases}$$

Proof. There are two cases depending on whether z is in e or not.

Case $z \in e$: We prove this case by simultaneous induction on the $pointsto(d,\mu)$, $pointsto(\overline{d},\mu)$, and $pointsto(e,\mu)$ rules.

Case POINTSTO-DEF*:* $pointsto([l/z](\texttt{def}\ m(x:\tau_1):\tau_2 = e'),\mu)$

$= pointsto(\texttt{def}\ m(x:\tau_1):\tau_2 = [l/z]e',\mu)$
$= pointsto([l/z]e',\mu)$ (POINTSTO-DEF)
$= pointsto(l,\mu) \cup pointsto(e',\mu)$ $(by\ IH)$
$= pointsto(l,\mu) \cup pointsto(\texttt{def}\ m(x:\tau_1):\tau_2 = e',\mu)$ (POINTSTO-DEF)

Case POINTSTO-VARX*:* Since there is only one variable, $x = z$.

$pointsto([l/z](\texttt{var}\ f:\tau = x),\mu)$
$= pointsto(\texttt{var}\ f:\tau = l,\mu)$
$= pointsto(l,\mu)$ (POINTSTO-VARL)
$= pointsto(l,\mu) \cup pointsto(\texttt{var}\ f:\tau = x,\mu)$ (POINTSTO-VARX)

Case POINTSTO-VARL*:* Since there are no variables, the substitution cannot take place, and the case is true by contradiction.

Case POINTSTO-DECLS*:* $pointsto([l/z]\overline{d}, \mu)$

$$= \bigcup_{d \in \overline{d}} pointsto([l/z]d, \mu) \quad \text{(POINTSTO-DECLS)}$$

$$= pointsto(l, \mu) \cup \bigcup_{d \in \overline{d}} pointsto(d, \mu) \quad \text{(POINTSTO-DEF, POINTSTO-VARX, POINTSTO-VARL)}$$

$$= pointsto(l, \mu) \cup pointsto(\overline{d}, \mu) \quad \text{(POINTSTO-DECLS)}$$

Case POINTSTO-VAR*:* Since there is only one variable, $x = z$.

$$pointsto([l/z]x, \mu) = pointsto(l, \mu) = pointsto(l, \mu) \cup pointsto(x, \mu) \quad \text{(POINTSTO-VAR)}$$

Case POINTSTO-NEW*:* $pointsto([l/z](\texttt{new}_s(x \Rightarrow \overline{d})), \mu)$

$$= pointsto(\texttt{new}_s(x \Rightarrow [l/z]\overline{d}), \mu)$$
$$= pointsto([l/z]\overline{d}, \mu) \quad \text{(POINTSTO-NEW)}$$
$$= pointsto(l, \mu) \cup pointsto(\overline{d}, \mu) \quad (by\ case\ \text{POINTSTO-DECLS})$$
$$= pointsto(l, \mu) \cup pointsto(\texttt{new}_s(x \Rightarrow \overline{d}), \mu) \quad \text{(POINTSTO-NEW)}$$

Case POINTSTO-METHOD*:* $pointsto([l/z](e.m(e')), \mu)$

$$= pointsto(([l/z]e).m([l/z]e'), \mu)$$
$$= pointsto([l/z]e, \mu) \cup pointsto([l/z]e', \mu) \quad \text{(POINTSTO-METHOD)}$$
$$= pointsto(l, \mu) \cup pointsto(e, \mu) \cup pointsto(e', \mu) \quad (by\ IH)$$
$$= pointsto(l, \mu) \cup pointsto(e.m(e'), \mu) \quad \text{(POINTSTO-METHOD)}$$

Case POINTSTO-FIELD*:* $pointsto([l/z](e.f), \mu)$

$$= pointsto(([l/z]e).f), \mu)$$
$$= pointsto([l/z]e, \mu) \quad \text{(POINTSTO-FIELD)}$$
$$= pointsto(l, \mu) \cup pointsto(e, \mu) \quad (by\ IH)$$
$$= pointsto(l, \mu) \cup pointsto(e.f, \mu) \quad \text{(POINTSTO-FIELD)}$$

Case POINTSTO-ASSIGN*:* $pointsto([l/z](e.f = e'), \mu)$

$$= pointsto(([l/z]e).f = [l/z]e', \mu)$$
$$= pointsto([l/z]e, \mu) \cup pointsto([l/z]e', \mu) \quad \text{(POINTSTO-ASSIGN)}$$
$$= pointsto(l, \mu) \cup pointsto(e, \mu) \cup pointsto(e', \mu) \quad (by\ IH)$$
$$= pointsto(l, \mu) \cup pointsto(e.f = e', \mu) \quad \text{(POINTSTO-ASSIGN)}$$

Case POINTSTO-BIND*:* $pointsto([l/z](\texttt{bind}\ x = e\ \texttt{in}\ e'), \mu)$

$$= pointsto(\texttt{bind}\ x = [l/z]e\ \texttt{in}\ [l/z]e', \mu)$$
$$= pointsto([l/z]e, \mu) \cup pointsto([l/z]e', \mu) \quad \text{(POINTSTO-BIND)}$$
$$= pointsto(l, \mu) \cup pointsto(e, \mu) \cup pointsto(e', \mu) \quad (by\ IH)$$
$$= pointsto(l, \mu) \cup pointsto(\texttt{bind}\ x = e\ \texttt{in}\ e', \mu) \quad \text{(POINTSTO-BIND)}$$

Case POINTSTO-PRINCIPAL *or* POINTSTO-PURE*:* Since there are no variables, the substitution cannot take place, and the case is true by contradiction.
Case POINTSTO-CALL-PRINCIPAL *or* POINTSTO-CALL-PURE*:* Since both the cases have method-call stack frames and the premise prohibits that, the cases are true by contradiction.

Case $z \notin e$*:* $[l/z]e = e$ and $pointsto([l/z]e, \mu) = pointsto(e, \mu)$. □

A.2.4 Capability-Safety Theorem

Theorem 6 (Capability Safety)**.** *If*

1. $\Gamma \mid \Sigma \vdash^{e'''} e : \tau$, [$e$ is well-typed]
2. $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$, [a step of evaluation is made]
3. $l_0 \mapsto \{x \Rightarrow \overline{d_0}\}_{\texttt{resource}} \in \mu'$, [$l_0$ is a principal]
4. $l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu$, *and* [$l$ is a principal]
5. $cap(l, e', \mu') \setminus cap(l, e, \mu) \supseteq \{l_0\}$,
   [between the two states, l's capability set increases by l_0]

then one of the following must be true:

1. ***Object creation:***
   (a) $e = E[l.m(l') \rhd E'[\texttt{new}_{\texttt{resource}}(x \Rightarrow \overline{d_0})]]$ *and* [a new principal was created
   (b) $e' = E[l.m(l') \rhd E'[l_0]]$, *where* in this evaluation step]
   (c) $\forall l_a.m_a(l'_a) \rhd E'' \in E'$, $l_a \mapsto \{x \Rightarrow \overline{d_a}\}_{\texttt{pure}} \in \mu$
   [there are only pure callers after the last method-call stack frame where l is the caller]
2. ***Method call:***
   (a) $e = E[l.m(l_0)]$, [a method argument was fully evaluated
   (b) $e' = E[l.m(l_0) \rhd [l_0/y][l/x]e'']$, *and* in this evaluation step]
   (c) $y \in e''$ [the passed-in argument y is used in the method body e'']
3. ***Method return:***
   (a) $e = E[l.m(l') \rhd E'[l_a.m_a(l'_a) \rhd l_0]]$ *and* [a method call returned
   (b) $e' = E[l.m(l') \rhd E'[l_0]]$, *where* in this evaluation step]
   (c) $\forall l_b.m_b(l'_b) \rhd E'' \in E'$, $l_b \mapsto \{x \Rightarrow \overline{d_b}\}_{\texttt{pure}} \in \mu$
   [there are only pure callers after the last method-call stack frame where l is the caller.]

Proof. The proof is by induction on a derivation of $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. For a given derivation, we proceed by cases on the last evaluation rule used:

Case E-CONGRUENCE: $\langle E[e] \mid \mu \rangle \longrightarrow \langle E[e'] \mid \mu' \rangle$
Let us enumerate method-call stack frames in E:
$E[e] = E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots]$
$E[e'] = E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e'] \ldots]$
where

1. for $1 \leq i \leq k$, $l'.m(l'') \rhd E' \notin E_i$ [no method-call stack frames in E_i]

2. $\forall i$, such that $l_i = l$, $i \in \{q_1, q_2, \ldots, q_{r_1}\}$, where $0 \leq r_1 \leq k$
   [the set of indices of all method-call stack frames where l is the caller; this set can be empty]
3. $\forall i \in \{q_1, q_2, \ldots, q_{r_1}\}$, if $\exists j$, such that
   (a) $l_j \mapsto \{x \Rightarrow \overline{d_j}\}_{\texttt{resource}} \in \mu$ and [l_j is a principal]
   (b) $\forall t$, such that $i > t > j$ and $l_t \mapsto \{x \Rightarrow \overline{d_t}\}_{\texttt{pure}} \in \mu$
   [all callers between l_i and l_j are pure]
   $j \in \{p_1, p_2, \ldots, p_{r_2}\}$ where $0 \leq r_2 \leq r_1$
   [the maximal set of indices of principal callers immediately after method-call stack frames where l is the caller; this set can be smaller than the one above only by one element; this set can also be empty; such principals can be l itself]

Then,
$cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e'] \ldots], \mu')$
$\setminus\ cap(l, E_k[l_k.m_k(l'_k) \rhd E_{k-1}[l_{k-1}.m_{k-1}(l'_{k-1}) \rhd \cdots \rhd E_2[l_2.m_2(l'_2) \rhd E_1[l_1.m_1(l'_1) \rhd e] \ldots], \mu)$

$$= \begin{cases} cap_{store}(l, \mu') \cup pointsto(e', \mu') \cup cap_{stack}(l, e', \mu') & \text{if } r_2 < r_1 \\ \setminus\ cap_{store}(l, \mu) \cup pointsto(e, \mu) \cup cap_{stack}(l, e, \mu) & \\ cap_{store}(l, \mu') \cup cap_{stack}(l, e', \mu') & \text{if } r_2 = r_1 \\ \setminus\ cap_{store}(l, \mu) \cup cap_{stack}(l, e, \mu) & \end{cases} \quad (Lemma\ 8)$$

[If $r_2 < r_1$, then there are only pure callers after the last method-call stack frame where l is the caller. In other words, l was the last principal caller on the stack.
If $r_2 = r_1$, then the last method-call stack frame where l is the caller is followed by a method-call stack frame with a principal caller that is not l. If $r_2 = r_1 = 0$, then there are no method-call stack frames with principal callers on the stack.
Since the set in 3(b) can include indices of method-call stack frames where the caller is l, the difference between r_1 and r_2 is at most 1, i.e., $r_2 \leq r_1 \leq r_2 + 1$.]
Thus, the changes in capabilities when $\langle E[e] \mid \mu\rangle \longrightarrow \langle E[e'] \mid \mu'\rangle$ depend on what expressions are in $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$. Let us consider all possible e and e'.

Subcase E-NEW: $e = \texttt{new}_s(x \Rightarrow \overline{d_a})$, $e' = l_a$, and $\langle E[\texttt{new}_s(x \Rightarrow \overline{d_a})] \mid \mu\rangle \longrightarrow \langle E[l_a] \mid \mu'\rangle$, where $\mu' = \mu, l_a \mapsto \{x \Rightarrow \overline{d_a}\}_s$.

By CAP-STORE, $cap_{store}(l, \mu) = pointsto(l, \mu) \cup pointsto(\overline{d}, \mu)$ and $cap_{store}(l, \mu') = pointsto(l, \mu') \cup pointsto(\overline{d}, \mu')$. By POINTSTO-PRINCIPAL and POINTSTO-PURE, $pointsto(l, \mu') = pointsto(l, \mu)$. By POINTSTO-DECLS and the $pointsto(d, \mu)$ rules, $pointsto(\overline{d}, \mu)$ depends only on what is in $\overline{d}$ and whether it is resource. Then, since the only change to the store was the addition of a new object l_a, and by inversion on E-NEW, $l_a \notin dom(\mu)$, and it is fully defined and all objects in $\overline{d_a}$ must be in the store at the time of the object creation (T-STORE), $pointsto(l_a, \mu) \notin pointsto(\overline{d}, \mu')$. Thus, $cap_{store}(l, \mu') = cap_{store}(l, \mu)$.

Case $r_2 < r_1$: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$= pointsto(l_a, \mu') \cup cap_{stack}(l, l_a, \mu')$

$\setminus\ pointsto(\texttt{new}_s(x \Rightarrow \overline{d_a}), \mu) \cup cap_{stack}(l, \texttt{new}_s(x \Rightarrow \overline{d_a}), \mu)$

$= pointsto(l_a, \mu')\ \setminus\ pointsto(\texttt{new}_s(x \Rightarrow \overline{d_a}), \mu)$ (CAP-STACK-NOCALL $\times$ 2)

$= pointsto(l_a, \mu')\ \setminus\ pointsto(\overline{d_a}, \mu)$ (POINTSTO-NEW)

There are two possibilities depending on whether l_a is a principal or not.

Case l_a is a principal:

$cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu) = \{l_a\}\ \setminus\ pointsto(\overline{d_a}, \mu)$ (POINTSTO-PRINCIPAL)

Since l_a points to a fresh memory location and our language requires an object to be allocated in memory before it can be used, $\{l_a\} \notin pointsto(\overline{d_a}, \mu)$, the capability set of l increases, which is in accordance with the ***object creation*** case, and the theorem holds.

Case l_a is pure: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu) = \varnothing\ \setminus\ pointsto(\overline{d_a}, \mu)$ (POINTSTO-PURE)

Thus, the capability set of l does not increase, and the theorem holds.

Case $r_2 = r_1$: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$= cap_{stack}(l, l_a, \mu')\ \setminus\ cap_{stack}(l, \texttt{new}_s(x \Rightarrow \overline{d_a}), \mu) = \varnothing$ (CAP-STACK-NOCALL $\times$ 2)

Thus, the capability set of l does not increase, and the theorem holds.

Subcase E-METHOD: $e = l_a.m(l_b)$, $e' = l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a$, $\mu' = \mu$, and $cap_{store}(l, \mu') = cap_{store}(l, \mu)$. Since e_a is a method definition, by Property 4, e_a has no method-call stack frames.

Case $r_2 < r_1$: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$= pointsto(l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu)$

$\setminus\ pointsto(l_a.m(l_b), \mu) \cup cap_{stack}(l, l_a.m(l_b), \mu)$

$= pointsto(l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu)$

$\setminus\ pointsto(l_a.m(l_b), \mu)$

(CAP-STACK-NOCALL)

There are three possibilities depending on whether $l_a = l$ and whether it is a principal or not.

Case $l_a = l$: Since l is a principal, l_a is a principal too.

$cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$= \{l_a\} \cup cap_{stack}(l, l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu)$ (POINTSTO-CALL-PRINCIPAL)

$\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu)$ (POINTSTO-METHOD)

$= \{l_a\} \cup pointsto([l_b/y][l_a/x]e_a, \mu) \cup cap_{stack}(l, [l_b/y][l_a/x]e_a, \mu)$ (CAP-STACK)

$\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu)$

$= \{l_a\} \cup pointsto([l_b/y][l_a/x]e_a, \mu)$ (CAP-STACK-NOCALL)

$\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu)$

$= pointsto(l_a, \mu) \cup pointsto([l_b/y][l_a/x]e_a, \mu)$ (POINTSTO-PRINCIPAL)

$\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu)$

$$
= \begin{cases}
pointsto(l_a, \mu) \cup pointsto(l_b, \mu) \cup pointsto(e_a, \mu) & \text{if } x, y \in e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) & \\
pointsto(l_a, \mu) \cup pointsto(e_a, \mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) & \\
pointsto(l_a, \mu) \cup pointsto(l_b, \mu) \cup pointsto(e_a, \mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) & \\
pointsto(l_a, \mu) \cup pointsto(e_a, \mu) & \text{if } x, y \notin e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) &
\end{cases}
$$

$(Lemma\ 9 \times 2)$

$$
= \begin{cases}
pointsto(e_a, \mu) & \text{if } x, y \in e_a \\
pointsto(e_a, \mu)\ \setminus\ pointsto(l_b, \mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\
pointsto(e_a, \mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\
pointsto(e_a, \mu)\ \setminus\ pointsto(l_b, \mu) & \text{if } x, y \notin e_a
\end{cases}
$$

$$
\subseteq pointsto(e_a, \mu)
$$

$$
= cap_{store}(l, \mu) \cup pointsto(e_a, \mu)\ \setminus\ cap_{store}(l, \mu)
$$

By CAP-STORE, POINTSTO-DECLS, and POINTSTO-DEF, $cap_{store}(l, \mu) \supseteq pointsto(e_a, \mu)$, and therefore, $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu) = \varnothing$. Thus, the capability set of l does not increase, and the theorem holds.

Case $l_a \neq l$ and l_a is a principal: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
= \{l_a\}\ \setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) \qquad \text{(POINTSTO-CALL-PRINCIPAL)}
$$

$$
= pointsto(l_a, \mu)\ \setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) = \varnothing \qquad \text{(POINTSTO-PRINCIPAL)}
$$

Thus, the capability set of l does not increase, and the theorem holds.

Case $l_a \neq l$ and l_a is pure: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
= pointsto([l_b/y][l_a/x]e_a, \mu)\ \setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) \quad \text{(POINTSTO-CALL-PURE)}
$$

$$
= \begin{cases}
pointsto(l_a, \mu) \cup pointsto(l_b, \mu) \cup pointsto(e_a, \mu) & \text{if } x, y \in e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) & \\
pointsto(l_a, \mu) \cup pointsto(e_a, \mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) & \\
pointsto(l_a, \mu) \cup pointsto(l_b, \mu) \cup pointsto(e_a, \mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) & \\
pointsto(l_a, \mu) \cup pointsto(e_a, \mu) & \text{if } x, y \notin e_a \\
\setminus\ pointsto(l_a, \mu) \cup pointsto(l_b, \mu) &
\end{cases}
$$

$(Lemma\ 9 \times 2)$

$$
= \begin{cases}
pointsto(e_a, \mu) & \text{if } x, y \in e_a \\
pointsto(e_a, \mu)\ \setminus\ pointsto(l_b, \mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\
pointsto(e_a, \mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\
pointsto(e_a, \mu)\ \setminus\ pointsto(l_b, \mu) & \text{if } x, y \notin e_a
\end{cases}
$$

$$
\subseteq pointsto(e_a, \mu)
$$

$$
= cap_{store}(l, \mu) \cup pointsto(e_a, \mu)\ \setminus\ cap_{store}(l, \mu)
$$

By CAP-STORE, POINTSTO-DECLS, and POINTSTO-DEF, $cap_{store}(l, \mu) \supseteq pointsto(e_a, \mu)$, and therefore, $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu) = \varnothing$. Thus, the capability set of l does not increase, and the theorem holds.

Case $r_2 = r_1$: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
= cap_{stack}(l, l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu)\ \setminus\ cap_{stack}(l, l_a.m(l_b), \mu)
$$

$$
= cap_{stack}(l, l_a.m(l_b) \rhd [l_b/y][l_a/x]e_a, \mu) \qquad \text{(CAP-STACK-NOCALL)}
$$

There are two possibilities depending on whether $l_a = l$ or not.

Case $l_a = l$: Since l is a principal, l_a is a principal too.
$cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
= pointsto([l_b/y][l_a/x]e_a, \mu) \cup cap_{stack}(l, [l_b/y][l_a/x]e_a, \mu) \qquad \text{(CAP-STACK)}
$$

$$
= pointsto([l_b/y][l_a/x]e_a, \mu) \qquad \text{(CAP-STACK-NOCALL)}
$$

$$= \begin{cases} pointsto(l_a,\mu) \cup pointsto(l_b,\mu) \cup pointsto(e_a,\mu) & \text{if } x,y \in e_a \\ pointsto(l_a,\mu) \cup pointsto(e_a,\mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\ pointsto(l_b,\mu) \cup pointsto(e_a,\mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\ pointsto(e_a,\mu) & \text{if } x,y \notin e_a \end{cases}$$

$(Lemma\ 9 \times 2)$

$$= \begin{cases} cap_{store}(l,\mu) \cup pointsto(l_a,\mu) & \text{if } x,y \in e_a \\ \cup pointsto(l_b,\mu) \cup pointsto(e_a,\mu) & \\ \setminus\ cap_{store}(l,\mu) & \\ cap_{store}(l,\mu) \cup pointsto(l_a,\mu) \cup pointsto(e_a,\mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\ \setminus\ cap_{store}(l,\mu) & \\ cap_{store}(l,\mu) \cup pointsto(l_b,\mu) \cup pointsto(e_a,\mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\ \setminus\ cap_{store}(l,\mu) & \\ cap_{store}(l,\mu) \cup pointsto(e_a,\mu)\ \setminus\ cap_{store}(l,\mu) & \text{if } x,y \notin e_a \end{cases}$$

Since $l_a = l$ and by CAP-STORE, POINTSTO-DECLS, and POINTSTO-DEF, $cap_{store}(l,\mu) \supseteq pointsto(l_a,\mu) \cup pointsto(e_a,\mu)$. Then, $cap(l,E[e'],\mu')\ \setminus\ cap(l,E[e],\mu)$

$$= \begin{cases} cap_{store}(l,\mu) \cup pointsto(l_b,\mu)\ \setminus\ cap_{store}(l,\mu) & \text{if } x,y \in e_a \\ cap_{store}(l,\mu)\ \setminus\ cap_{store}(l,\mu) & \text{if } x \in e_a \text{ and } y \notin e_a \\ cap_{store}(l,\mu) \cup pointsto(l_b,\mu)\ \setminus\ cap_{store}(l,\mu) & \text{if } x \notin e_a \text{ and } y \in e_a \\ cap_{store}(l,\mu)\ \setminus\ cap_{store}(l,\mu) & \text{if } x,y \notin e_a \end{cases}$$

$$= \begin{cases} pointsto(l_b,\mu) & \text{if } y \in e_a \\ \varnothing & \text{if } y \notin e_a \end{cases}$$

$$= \begin{cases} \{l_b\} & \text{if } y \in e_a \text{ and } l_b \text{ is a principal} \quad (\text{POINTSTO-PRINCIPAL}) \\ \varnothing & \text{otherwise} \end{cases}$$

Thus, if $y \in e_a$ and l_b is a principal, the capability set of l increases, which is in accordance with the ***method call*** case, and the theorem holds.

Case $l_a \neq l$: $cap(l,E[e'],\mu')\ \setminus\ cap(l,E[e],\mu) = \varnothing$ (CAP-STACK-NOCALL)

Thus, the capability set of l does not increase, and the theorem holds.

Subcase E-FIELD: $e = l_a.f$, $e' = l_b$, $\mu' = \mu$, and $cap_{store}(l,\mu') = cap_{store}(l,\mu)$.

By Property 5, the object field that is being accessed must belong to the caller of the last method-call stack frame on the stack. Then, $l_1 = l_a$. Considering that e is well-typed, since l_1 has a field, by definition, l_1 is a principal.

Case $r_2 < r_1$: Since l_1 is a principal, $l = l_1 = l_a$.

$cap(l,E[e'],\mu')\ \setminus\ cap(l,E[e],\mu)$

$= pointsto(l_b, \mu) \cup cap_{stack}(l, l_b, \mu) \setminus pointsto(l.f, \mu) \cup cap_{stack}(l, l.f, \mu)$
$= pointsto(l_b, \mu) \setminus pointsto(l.f, \mu)$ (CAP-STACK-NOCALL $\times 2$)
$= cap_{store}(l, \mu) \cup pointsto(l_b, \mu) \setminus cap_{store}(l, \mu) \cup pointsto(l.f, \mu)$

By inversion on E-FIELD, $\texttt{var}\ f : \tau = l_b \in \overline{d}$. Then, by CAP-STORE, POINTSTO-DECLS, and POINTSTO-VARL, $cap_{store}(l, \mu) \supseteq pointsto(l_b, \mu)$, and

$$cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu) = cap_{store}(l, \mu) \setminus cap_{store}(l, \mu) \cup pointsto(l.f, \mu) = \varnothing$$

Thus, the capability set of l does not increase, and the theorem holds.

Case $r_2 = r_1$: $cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu)$
$= cap_{stack}(l, l_b, \mu) \setminus cap_{stack}(l, l_a.f, \mu) = \varnothing$ (CAP-STACK-NOCALL $\times 2$)

Thus, l's capability set does not increase, and the theorem holds.

Subcase E-ASSIGN: $e = (l_a.f = l_b)$, $e' = l_b$, and by inversion on E-ASSIGN, $l_a \mapsto \{x \Rightarrow \overline{d_a}\}_s \in \mu$, $\texttt{var}\ f : \tau = l_c \in \overline{d_a}$, $\overline{d_a}' = [\texttt{var}\ f : \tau = l_b / \texttt{var}\ f : \tau = l_c]\overline{d_a}$, and $\mu' = [l_a \mapsto \{x \Rightarrow \overline{d_a}'\}_s / l_a \mapsto \{x \Rightarrow \overline{d_a}\}_s]\mu$.

By Property 5, the object field that is being accessed must belong to the caller of the last method-call stack frame on the stack. Then, $l_1 = l_a$. Considering that e is well-typed, since l_1 has a field, by definition, l_1 is a principal.

Case $r_2 < r_1$: Since l_1 is a principal, in this case, $l = l_1 = l_a$.

Since in this step of evaluation, the only change to the store is the substitution of l_c with l_b in one of l's fields, by CAP-STORE, POINTSTO-DECLS, and POINTSTO-VARL, $cap_{store}(l, \mu') \setminus cap_{store}(l, \mu) \subseteq pointsto(l_b, \mu')$. [1]

By POINTSTO-PRINCIPAL and POINTSTO-PURE, $pointsto(l_b, \mu') = pointsto(l_b, \mu)$. [2]

$cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu)$
$= cap_{store}(l, \mu') \cup pointsto(l_b, \mu') \cup cap_{stack}(l, l_b, \mu')$
$\setminus cap_{store}(l, \mu) \cup pointsto(l.f = l_b, \mu) \cup cap_{stack}(l, l.f = l_b, \mu)$
$= cap_{store}(l, \mu') \cup pointsto(l_b, \mu') \setminus cap_{store}(l, \mu) \cup pointsto(l.f = l_b, \mu)$
(CAP-STACK-NOCALL $\times 2$)
$= cap_{store}(l, \mu') \cup pointsto(l_b, \mu') \setminus cap_{store}(l, \mu) \cup pointsto(l, \mu) \cup pointsto(l_b, \mu)$
(POINTSTO-ASSIGN)
$\subseteq pointsto(l_b, \mu') \setminus pointsto(l, \mu) \cup pointsto(l_b, \mu)$ (*by* [1])
$= \varnothing$ (*by* [2])

Thus, the capability set of l does not increase, and the theorem holds.

Case $r_2 = r_1$: Since l_1 is a principal and $l_1 = l_a$, in this case, $l \neq l_a$ and $r_2 = r_1 \neq 0$.

Since $l \neq l_a$ and, in this step of evaluation, the only change to the store is the substitution of l_c with l_b in one of l_1's fields, by CAP-STORE, POINTSTO-DECLS, and POINTSTO-VARL, $cap_{store}(l, \mu') = cap_{store}(l, \mu)$. [3]

$$
\begin{aligned}
&cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu) \\
&\quad = cap_{store}(l, \mu') \cup cap_{stack}(l, l_b, \mu') \setminus cap_{store}(l, \mu) \cup cap_{stack}(l, l_a.f = l_b, \mu) \\
&\quad = cap_{store}(l, \mu') \setminus cap_{store}(l, \mu) \\
&\quad (\text{CAP-STACK-NOCALL} \times 2) \\
&\quad = \varnothing \qquad (by\ [3])
\end{aligned}
$$

Thus, the capability set of l does not increase, and the theorem holds.

Subcase E-BIND: $e = \texttt{bind}\ x = l_a\ \texttt{in}\ e_a$, $e' = [l_a/x]e_a$, $\mu' = \mu$, and $cap_{store}(l, \mu') = cap_{store}(l, \mu)$. Since e_a is a method definition, by Property 4, e_a has no method-call stack frames.

Case $r_2 < r_1$: $cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu)$

$$
\begin{aligned}
&= pointsto([l_a/x]e_a, \mu) \cup cap_{stack}(l, [l_a/x]e_a, \mu) \\
&\setminus pointsto(\texttt{bind}\ x = l_a\ \texttt{in}\ e_a, \mu) \cup cap_{stack}(l, \texttt{bind}\ x = l_a\ \texttt{in}\ e_a, \mu) \\
&= pointsto([l_a/x]e_a, \mu) \setminus pointsto(\texttt{bind}\ x = l_a\ \texttt{in}\ e_a, \mu) \\
&(\text{CAP-STACK-NOCALL} \times 2) \\
&= pointsto([l_a/x]e_a, \mu) \setminus pointsto(l_a, \mu) \cup pointsto(e_a, \mu) \qquad (\text{POINTSTO-BIND}) \\
&= \begin{cases} pointsto(l_a, \mu) \cup pointsto(e_a, \mu) \setminus pointsto(l_a, \mu) \cup pointsto(e_a, \mu) & \text{if } x \in e_a \\ pointsto(e_a, \mu) \setminus pointsto(l_a, \mu) \cup pointsto(e_a, \mu) & \text{if } x \notin e_a \end{cases} \\
&(Lemma\ 9) \\
&= \varnothing
\end{aligned}
$$

Thus, the capability set of l does not increase, and the theorem holds.

Case $r_2 = r_1$: $cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu)$

$$
\begin{aligned}
&= cap_{stack}(l, [l_a/x]e_a, \mu) \setminus cap_{stack}(l, \texttt{bind}\ x = l_a\ \texttt{in}\ e_a, \mu) \\
&= \varnothing \qquad (\text{CAP-STACK-NOCALL} \times 2)
\end{aligned}
$$

Thus, l's capability set does not increase, and the theorem holds.

Subcase E-STACKFRAME: $e = l_a.m(l_b) \rhd l_c$, $e' = l_c$, $\mu' = \mu$, and $cap_{store}(l, \mu') = cap_{store}(l, \mu)$.

Case $r_2 < r_1$: $cap(l, E[e'], \mu') \setminus cap(l, E[e], \mu)$

$$
\begin{aligned}
&= pointsto(l_c, \mu) \cup cap_{stack}(l, l_c, \mu) \\
&\setminus\ pointsto(l_a.m(l_b) \rhd l_c, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&= pointsto(l_c, \mu)\ \setminus\ pointsto(l_a.m(l_b) \rhd l_c, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&\text{(CAP-STACK-NOCALL)}
\end{aligned}
$$

There are three possibilities depending on whether $l_a = l$ and whether it is a principal or not.
Case $l_a = l$: Since l is a principal, l_a is a principal too.

$$
\begin{aligned}
&cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu) \\
&\quad = pointsto(l_c, \mu)\ \setminus\ pointsto(l_a.m(l_b) \rhd l_c, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&\quad = pointsto(l_c, \mu) && \text{(CAP-STACK)} \\
&\quad \setminus\ pointsto(l_a.m(l_b) \rhd l_c, \mu) \cup pointsto(l_c, \mu) \cup cap_{stack}(l, l_c, \mu) \\
&\quad = \varnothing
\end{aligned}
$$

Thus, the capability set of l does not increase, and the theorem holds.
Case $l_a \neq l$ and l_a is a principal: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
\begin{aligned}
&= pointsto(l_c, \mu)\ \setminus\ pointsto(l_a.m(l_b) \rhd l_c, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&= pointsto(l_c, \mu)\ \setminus\ \{l_a\} \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) && \text{(POINTSTO-CALL-PRINCIPAL)} \\
&= pointsto(l_c, \mu)\ \setminus\ \{l_a\} && \text{(CAP-STACK-NOCALL)} \\
&= \begin{cases} \{l_c\}\ \setminus\ \{l_a\} & \text{if } l_c \text{ is a principal} \quad \text{(POINTSTO-PRINCIPAL)} \\ \varnothing & \text{if } l_c \text{ is pure} \qquad \text{(CAP-STACK-NOCALL)} \end{cases}
\end{aligned}
$$

Thus, if $l_a \neq l$, l_a is a principal, and l_c is a principal, then the capability set of l increases, which is in accordance with the ***method return*** case, and the theorem holds. If $l_a \neq l$, l_a is a principal, and l_c is pure, then the capability set of l does not increase, and the theorem holds.
Case $l_a \neq l$ and l_a is pure: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
\begin{aligned}
&= pointsto(l_c, \mu)\ \setminus\ pointsto(l_a.m(l_b) \rhd l_c, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&= pointsto(l_c, \mu)\ \setminus\ pointsto(l_c, \mu) \cup cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&= \varnothing\ \setminus\ cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) && \text{(POINTSTO-CALL-PURE)}
\end{aligned}
$$

Thus, the capability set of l does not increase, and the theorem holds.

Case $r_2 = r_1$: $cap(l, E[e'], \mu')\ \setminus\ cap(l, E[e], \mu)$

$$
\begin{aligned}
&= cap_{stack}(l, l_c, \mu)\ \setminus\ cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) \\
&= \varnothing\ \setminus\ cap_{stack}(l, l_a.m(l_b) \rhd l_c, \mu) = \varnothing && \text{(CAP-STACK-NOCALL)}
\end{aligned}
$$

Thus, l's capability set does not increase, and the theorem holds. □

Appendix B

Authority Safety via Effects

B.1 Type Soundness

B.1.1 Lemmas

Lemma 10 (Permutation)**.** *If* $\Gamma \mid \varnothing \vdash e : \{\varepsilon\}\ \tau$ *and* Δ *is a permutation of* Γ*, then* $\Delta \mid \varnothing \vdash e : \{\varepsilon\}\ \tau$*, and the latter derivation has the same depth as the former.*

Proof. Straightforward induction on typing derivations. □

Lemma 11 (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 12 (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*:* $\tau_1 = \{x \Rightarrow \sigma_i^{i \in 1..n}\}_s$ and $\tau_2 = \{x \Rightarrow \sigma_i'^{i \in 1..n}\}_s$. Substitution preserves the permutation relations, and thus, $[l/z]\{x \Rightarrow \sigma_i^{i \in 1..n}\}_s$ is a permutation of $[l/z]\{x \Rightarrow \sigma_i'^{i \in 1..n}\}_s$. Then, by S-PERM, $\Gamma \vdash [l/z]\{x \Rightarrow \sigma_i^{i \in 1..n}\}_s <: [l/z]\{x \Rightarrow \sigma_i'^{i \in 1..n}\}_s$.

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

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

Case S-RESOURCE*:* $\tau_1 = \{x \Rightarrow \overline{\sigma}\}_{\texttt{pure}}$ and $\tau_2 = \{x \Rightarrow \overline{\sigma}\}_{\texttt{resource}}$, and the desired result is immediate.

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 $\Gamma,\ z : \tau \vdash \tau'_1 <: \tau_1$, $\Gamma,\ z : \tau \vdash \tau_2 <: \tau'_2$, $\mathit{lookup}((\Gamma,\ z : \tau), \varepsilon_1) = \varepsilon'_1$, $\mathit{lookup}((\Gamma,\ z : \tau), \varepsilon_2) = \varepsilon'_2$, and $\varepsilon'_1 \subseteq \varepsilon'_2$. By the induction hypothesis, $\Gamma \vdash [l/z]\tau'_1 <: [l/z]\tau_1$ and $\Gamma \vdash [l/z]\tau_2 <: [l/z]\tau'_2$. Substitution does not affect the relationship between effects, and thus, $\mathit{lookup}(\Gamma, [l/z]\varepsilon_1) = [l/z]\varepsilon'_1$, $\mathit{lookup}(\Gamma, [l/z]\varepsilon_2) = [l/z]\varepsilon'_2$, and $[l/z]\varepsilon'_1 \subseteq [l/z]\varepsilon'_2$. 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 13 (Substitution in expressions)**.** *If* $\Gamma,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ *and* $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$*, then* $\Gamma \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon\}\ [l/z]\tau$. *Furthermore, if* $\Gamma,\ z : \tau' \mid \Sigma \vdash_s d : \sigma$ *and* $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$*, then* $\Gamma \mid \Sigma \vdash_s [l/z]d : [l/z]\sigma$.

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

Case T-METHOD*:* $e = e_1.m(e_2)$, and by inversion on T-METHOD, we get $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$; $\texttt{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$;

$\Gamma,\ z : \tau' \mid \Sigma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf$; and $\Gamma,\ z : \tau' \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}_s$, $\texttt{def}\ m(y : [l/z]\tau_2) : \{[l/z]\varepsilon_3\}\ [l/z]\tau_1 \in [l/z]\overline{\sigma}$, $\Gamma \mid \Sigma \vdash [l/z]([e_1/x][e_2/y]\varepsilon_3)\ wf$, 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 $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}_s$ and $\texttt{var}\ f : \tau \in \overline{\sigma}$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}_s$ 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 $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$; $\texttt{var}\ f : \tau \in \overline{\sigma}$; and $\Gamma,\ z : \tau' \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ \tau$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}_s$; $\texttt{var}\ f : [l/z]\tau \in [l/z]\overline{\sigma}$; 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 $\Gamma,\ z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \tau_1$; $\Gamma \vdash \tau_1 <: \tau_2$; $\mathit{lookup}((\Gamma,\ z : \tau'), \varepsilon_1) = \varepsilon_1'$; $\mathit{lookup}((\Gamma,\ z : \tau'), \varepsilon_2) = \varepsilon_2'$; and $\varepsilon_1' \subseteq \varepsilon_2'$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\tau_1$. By Lemma 12 (substitution in types), $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_2$. Substitution does not affect the relationship between effects, and thus, $\mathit{lookup}(\Gamma, [l/z]\varepsilon_1) = [l/z]\varepsilon_1'$, $\mathit{lookup}(\Gamma, [l/z]\varepsilon_2) = [l/z]\varepsilon_2'$ and $[l/z]\varepsilon_1' \subseteq [l/z]\varepsilon_2'$. Then, by T-SUB, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_2\}\ [l/z]\tau_2$.

Case DT-DEFPURE: $d = \texttt{def}\ m(y : \tau_1) : \tau_2 = e$. By inversion on DT-DEFPURE, we get $\Gamma_{\texttt{resource}} = \{x : \{x \Rightarrow \overline{\sigma}\}_{\texttt{resource}} \mid x : \{x \Rightarrow \overline{\sigma}\}_{\texttt{resource}} \in \Gamma\}$; $\Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}}$; $\Gamma_{\texttt{pure}},\ y : \tau_1,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$; $\Gamma,\ y : \tau_1,\ z : \tau' \mid \Sigma \vdash \varepsilon\ wf$; and $\varepsilon \supseteq \varepsilon'$, and by the induction hypothesis, $\Gamma_{\texttt{pure}},\ y : [l/z]\tau_1 \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon'\}\ [l/z]\tau_2$; $\Gamma,\ y : [l/z]\tau_1 \mid \Sigma \vdash [l/z]\varepsilon\ wf$; and $[l/z]\varepsilon \supseteq [l/z]\varepsilon'$. Then, by DT-DEFPURE, $\Gamma \mid \Sigma \vdash_{\texttt{pure}} \texttt{def}\ m(y : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2 = [l/z]e : \texttt{def}\ m(y : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2$, i.e., $\Gamma \mid \Sigma \vdash_{\texttt{pure}} [l/z](\texttt{def}\ m(y : \tau_1) : \{\varepsilon\}\ \tau_2 = e)\ :\ [l/z](\texttt{def}\ m(y : \tau_1) : \{\varepsilon\}\ \tau_2)$.

Thus, in both cases, the type of d is preserved under substitution.

Case DT-DEFRESOURCE: $d = \texttt{def}\ m(x : \tau_1) : \tau_2 = e$, and by inversion on DT-DEFRESOURCE, we get $\Gamma, x : \tau_1,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$; $\Gamma,\ x : \tau_1,\ z : \tau' \mid \Sigma \vdash \varepsilon\ wf$; and $\varepsilon \supseteq \varepsilon'$. By the induction hypothesis, $\Gamma, x : [l/z]\tau_1 \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon'\}\ [l/z]\tau_2$; $\Gamma,\ x : [l/z]\tau_1 \mid \Sigma \vdash [l/z]\varepsilon\ wf$; and $[l/z]\varepsilon \supseteq [l/z]\varepsilon'$. Then, by DT-DEFRESOURCE, $\Gamma \mid \Sigma \vdash_{\texttt{resource}} \texttt{def}\ m(x : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2 = [l/z]e : \texttt{def}\ m(x : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2$, i.e., $\Gamma \mid \Sigma \vdash_{\texttt{resource}}\ [l/z](\texttt{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2 = e)\ :\ [l/z](\texttt{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2)$.

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, $\Gamma \mid \Sigma \vdash_{\texttt{resource}} \texttt{var}\ f : [l/z]\tau = l\ :\ \texttt{var}\ f : [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash_{\texttt{resource}} [l/z](\texttt{var}\ f : \tau = l)\ :\ [l/z](\texttt{var}\ f : \tau)$, 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, $\Gamma \mid \Sigma \vdash_{\texttt{resource}} \texttt{var}\ f : [l/z]\tau = x\ :\ \texttt{var}\ f : [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash_{\texttt{resource}} [l/z](\texttt{var}\ f : \tau = x)\ :\ [l/z](\texttt{var}\ f : \tau)$, 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.

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

B.1.2 Preservation

Theorem 7 (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* $lookup(\Gamma, \varepsilon') \subseteq lookup(\Gamma, \varepsilon)$*, 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*:* $e = \texttt{new}_s(x \Rightarrow \overline{d})$, and by inversion on T-NEW, we get $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\}_s \mid \Sigma \vdash_s d_i : \sigma_i$. The store changes from μ to $\mu' = \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s$, 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 $\mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s\ :\ \Sigma,\ l : \{x \Rightarrow \overline{\sigma}\}_s$, which implies that $\Sigma' = \Sigma,\ l : \{x \Rightarrow \overline{\sigma}\}_s$. Finally, by T-LOC, $\Gamma \mid \Sigma \vdash l : \{\}\ \{x \Rightarrow \overline{\sigma}\}_s$, 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 $\Gamma \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$, $\texttt{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma},\ \Gamma \mid \Sigma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf,\ \Gamma \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$, 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,

$\Gamma \mid \Sigma \vdash_s \texttt{def}\ m(x:\tau_1):\{\varepsilon\}\ \tau_2 = e\ :\ \texttt{def}\ m(x:\tau_1):\{\varepsilon\}\ \tau_2$. Then, by inversion on both DT-DEFPURE and DT-DEFRESOURCE, we know that $\Gamma, x:\tau_1 \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$ and $\mathit{lookup}(\Gamma, \varepsilon') \subseteq \mathit{lookup}(\Gamma, \varepsilon)$. Finally, by the subsumption lemma, substituting locations for variables in e preserve 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 $\Gamma \mid \Sigma \vdash l : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}_s$, 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, $\Gamma \mid \Sigma \vdash_s \texttt{var}\ \ f:\tau = l_1\ :\ \texttt{var}\ \ f:\tau$. 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 $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$, $\texttt{var}\ \ f:\tau \in \overline{\sigma}$, $\Gamma \mid \Sigma \vdash l_2 : \{\varepsilon_2\}\ \tau$, and $\varepsilon = \varepsilon_1 = \varepsilon_2 = \varnothing$. The store changes as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}_s / l_1 \mapsto \{x \Rightarrow \overline{d}\}_s]\mu$, 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, $\Gamma \mid \Sigma \vdash_s \texttt{var}\ \ f:\tau = l_2\ :\ \texttt{var}\ \ f:\tau$. 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.1.3 Progress

Theorem 8 (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*:* $e = \texttt{new}_s(x \Rightarrow \overline{d})$, 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 $\mu, l \mapsto \{x \Rightarrow \overline{d}\}_s$, 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 $\Gamma \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$, 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 $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$ 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 $l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ 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 $\Gamma \mid \Sigma \vdash l : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}_s$ 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 $l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ 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 $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}_s$, $\texttt{var}\ \ f : \tau \in \overline{\sigma}$, 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 $l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu$ with $\texttt{var}\ f : \tau = l \in \overline{d}$. A new well-typed store can be created as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d'}\}_s / l_1 \mapsto \{x \Rightarrow \overline{d}\}_s]\mu$, 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. □

B.2 More General Definition of Authority Attenuation

Definition 3 (Authority Attenuation (more generally))**.** Objects of type τ_1 attenuate objects of type τ_2, if

1. $F_1 = tLookup(\Gamma, \tau, auth(\tau_1))$, $F_2 = tLookup(\Gamma, \tau, auth(\tau_2))$,
2. $F_1 \cap F_2 \neq \varnothing$, and
3. $F_2 \setminus F_1 \neq \varnothing$.

This definition essentially says that if we let F_1 be the set of effects that represents authority of objects of one type and F_2 be the set of effects that represents authority of objects of another type. Then, if F_1 and F_2 share at least one effect and there is at least one effect that is in F_2 but not in F_1, we say that objects of the former type attenuate objects of the latter type.

Bibliography

[1] The Monte Programming Language. http://monte.readthedocs.io. 2.9

[2] Draft Proposal for SES (Secure EcmaScript). https://github.com/tc39/proposal-ses. 4e, 5.2

[3] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of Path-Dependent Types. *SIGPLAN Notices*, 49(10), October 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660216. URL https://doi.org/10.1145/2714064.2660216. 3.3.2

[4] Dor Azouri. Abusing Text Editors with Third-party Plugins. https://go.safebreach.com/rs/535-IXZ-934/images/Abusing_Text_Editors.pdf, 2018. 3.1

[5] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108 – 123, 2015. ISSN 2352-2208. doi: http://dx.doi.org/10.1016/j.jlamp.2014.02.001. URL http://www.sciencedirect.com/science/article/pii/S2352220814000194. 3.8

[6] Dariusz Biernacki, Maciej Piròg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting Algebraic Effects. In *Symposium on Principles of Programming Languages*, 2019. 3.8

[7] 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. 3

[8] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *European Conference on Object-Oriented Programming*, 2010. 1, 2.2, 2.4, 2.9

[9] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *International Symposium on Scala*, 2017. ISBN 978-1-4503-5529-2. doi: 10.1145/3136000.3136007. URL http://doi.acm.org/10.1145/3136000.3136007. 3.8

[10] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *International Conference on Functional Programming*, 2013. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500581. URL http://doi.acm.org/10.1145/2500365.2500581. 3.8

[11] 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`. 3, 3.8, 3.8

[12] Shuo Chen, David Ross, and Yi-Min Wang. An Analysis of Browser Domain-isolation Bugs and a Light-weight Transparent Defense Mechanism. In *Conference on Computer and Communications Security*, 2007. ISBN 978-1-59593-703-2. 1

[13] Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Evaluating the Flexibility of the Java Sandbox. In *Annual Computer Security Applications Conference*, 2015. 1, 2.6.1, 4.1

[14] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. Capabilities: Effects for Free. In *Formal Methods and Software Engineering*, 2018. ISBN 978-3-030-02450-5. 3.7

[15] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966. 2.6

[16] Dominique Devriese, Frank Piessens, and Lars Birkedal. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*, 2016. 1, 2.9, 3.8

[17] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium*, 2014. 2.9, 3.8, 5.1

[18] 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`. 3, 3.8

[19] Sophia Drossopoulou and James Noble. The Need for Capability Policies. In *Workshop on Formal Techniques for Java-like Programs*, 2013. 2.9

[20] Sophia Drossopoulou and James Noble. How to Break the Bank: Semantics of Capability Policies. In *Integrated Formal Methods*, 2014. 2.9

[21] Sophia Drossopoulou and James Noble. Towards Capability Policy Specification and Verification. Technical report, Victoria University of Wellington, 2014. 2.9

[22] Sophia Drossopoulou, James Noble, and Mark S. Miller. Swapsies on the Internet: First Steps Towards Reasoning About Risk and Trust in an Open World. In *Workshop on Programming Languages and Analysis for Security*, 2015. 2.9

[23] Sophia Drossopoulou, James Noble, Toby Murray, and Mark S. Miller. Reasoning about Risk and Trust in an Open World. Technical report, Victoria University of Wellington, 2015. 2.9

[24] Sophia Drossopoulou, James Noble, Mark S. Miller, and Toby Murray. Permission and Authority Revisited Towards a Formalisation. In *Workshop on Formal Techniques for Java-like Programs*, 2016. ISBN 978-1-4503-4439-5. 3.5

[25] Matthew Flatt and Matthias Felleisen. Units: Cool Modules for HOT Languages. In *Pro-*

gramming Language Design and Implementation, 1998. 2.9

[26] Google, Inc. Caja. https://code.google.com/p/google-caja/. 2.9

[27] Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. Capabilities for Java: Secure Access to Resources. In *Asian Symposium on Programming Languages and Systems*, 2017. ISBN 978-3-319-71237-6. doi: 10.1007/978-3-319-71237-6_4. URL https://doi.org/10.1007/978-3-319-71237-6_4. 2.9

[28] Michael Homer, Kim B. Bruce, James Noble, and Andrew P. Black. Modules As Gradually-typed Objects. In *Workshop on Dynamic Languages and Applications*, 2013. 1

[29] 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. 3, 3.7

[30] George Kuan and David MacQueen. Engineering Higher-Order Modules in SML/NJ. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages*, pages 218–235, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 1

[31] Ben Laurie. Safer Scripting Through Precompilation. In *Security Protocols*, 2007. 2.9

[32] 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/. 3.8

[33] 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. 3.8

[34] Sam Lindley, Conor McBride, and Craig McLaughlin. Do Be Do Be Do. In *Symposium on Principles of Programming Languages*, 2017. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009897. URL http://doi.acm.org/10.1145/3009837.3009897. 3.8

[35] Shu-Peng Loh and Sophia Drossopoulou. Specifying Attenuation. https://2017.splashcon.org/event/ocap-2017-specifying-attenuation, 2017. 3.8

[36] Justin Lubin. Approximating Polymorphic Effects with Capabilities. In *SPLASH 2018 Student Research Competition*, 2018. 3.7, 4.2.3, 4.2.3

[37] John M. Lucassen. *Types and Effects towards the Integration of Functional and Imperative Programming*. PhD thesis, Massachusetts Institute of Technology, 1987. 1.1, 3.8

[38] 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. 3, 3.8

[39] Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016. 1, 2.6.1, 4.1

[40] David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, 1984. 2.2, 2.2

[41] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy*, 2010. 1, 2.9, 3, 3.8, 5.1

[42] Daniel Marino and Todd Millstein. A Generic Type-and-effect System. In *International Workshop on Types in Language Design and Implementation*, 2009. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481868. URL `http://doi.acm.org/10.1145/1481861.1481868`. 3.8

[43] 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. 2.6.2

[44] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A Capability-Based Module System for Authority Control. Technical Report CMU-ISR-17-106, Carnegie Mellon University, 2017. URL `http://reports-archive.adm.cs.cmu.edu/anon/isr2017/abstracts/17-106.html`. 2.6.2

[45] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*, 2010. 2.9, 3.8, 5.1

[46] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *European Conference on Object-Oriented Programming*, 2014. 2.5.1

[47] Mark S. Miller. The E Language. `http://erights.org/elang/`. 2.4, 2.9

[48] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006. 1.1, 2.9, 3.3.4, 3.5, 3.5.3, 3.8, 3.8, 5.1

[49] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe Active Content in Sanitized JavaScript. Technical report, Google, Inc., 2008. 2.9

[50] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A Secure Shell Scripting Language. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014. 2.9

[51] Toby Murray. Analysing Object-Capability Security. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, 2008. 3.3.4

[52] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A Simple, Typed, and Pure Object-Oriented Language. In *Workshop on Mechanisms for Specialization, Generalization and Inheritance*, 2013. 1.1, 2.5, 5.1

[53] James Noble and Sophia Drossopoulou. Rationally Reconstructing the Escrow Example. In *Workshop on Formal Techniques for Java-like Programs*, 2014. 2.9

[54] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp

Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. Scala Language Specification. http://scala-lang.org/files/archive/spec/2.11/. Last accessed: May 2017. 1

[55] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP'14, pages 105–130. Springer, 2014. ISBN 978-3-662-44201-2. doi: 10.1007/978-3-662-44202-9_5. URL http://dx.doi.org/10.1007/978-3-662-44202-9_5. 3

[56] *Class SecurityManager*. Oracle Corporation. https://docs.oracle.com/javase/10/docs/api/java/lang/SecurityManager.html. 5.2

[57] David L. Parnas. Information Distribution Aspects of Design Methodology. volume 71, pages 339–344, 01 1971. 3.3.2

[58] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL http://doi.acm.org/10.1145/361598.361623. 3.3.2

[59] 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. 3.8

[60] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *Programming Languages and Systems*, 2009. ISBN 978-3-642-00590-9. 3.8

[61] Vineet Rajani, Deepak Garg, and Tamara Rezk. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 150–163, June 2016. doi: 10.1109/CSF.2016.18. 2.9

[62] Jonathan A. Rees. A Security Kernel Based on the Lambda-Calculus. Technical report, Massachusetts Institute of Technology, 1996. 2.9

[63] John M. Rushby. Design and Verification of Secure Systems. In *Symposium on Operating Systems Principles*, 1981. ISBN 0-89791-062-1. 1

[64] 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. 3.7, 3.8

[65] Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974. 1

[66] Z. Cliffe Schreuders, Tanya Mcgill, and Christian Payne. The State of the Art of Application Restrictions and Sandboxes: A Survey of Application-oriented Access Controls and Their Shortfalls. *Computers and Security*, 32:219–241, 2013. ISSN 0167-4048. 1

[67] Fred Spiessens and Peter Van Roy. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Multiparadigm Programming in Mozart/Oz*, 2005. 2.9

[68] Marc Stiegler. Emily: A High Performance Language for Enabling Secure Cooperation. In *International Conference on Creating, Connecting and Collaborating through Computing*, 2007. 2.9

[69] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1046. URL `http://dx.doi.org/10.1006/inco.1994.1046`. 5.2

[70] Mike Ter Louw, Prithvi Bisht, and V Venkatakrishnan. Analysis of Hypertext Isolation Techniques for XSS Prevention. *Web 2.0 Security and Privacy*, 2008. 1

[71] Franklyn A. Turbak and David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008. ISBN 0262201755, 9780262201759. 3, 3.8

[72] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: Programming Responsive Mobile Peer-to-peer Applications with Actors. *Computer Languages, Systems and Structures*, 40(34):112–136, 2014. 2.5.1

[73] David Wagner and Dean Tribble. A Security Analysis of the Combex DarpaBrowser Architecture. `http://combex.com/papers/darpa-review/security-review.pdf`, March 2002. 2.9, 5.1

[74] Esther Wang and Jonathan Aldrich. Capability Safe Reflection for the Wyvern Language. In *Workshop on Meta-Programming Techniques and Reflection*, 2016. 2.4

[75] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *USENIX Workshop on Offensive Technologies*, 2007. 1

[76] 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`. 3.8