Gradient: Gradual Compartmentalization via Object Capabilities Tracked in Types

ALEKSANDER BORUCH-GRUSZECKI, Charles University, Czechia
ADRIEN GHOSN, Azure Research, Microsoft, UK
MATHIAS PAYER, EPFL, Switzerland
CLÉMENT PIT-CLAUDEL, EPFL, Switzerland

Modern software needs fine-grained compartmentalization, i.e., intra-process isolation. A particularly important reason for it are supply-chain attacks, the need for which is aggravated by modern applications depending on hundreds or even thousands of libraries. Object capabilities are a particularly salient approach to compartmentalization, but they require the entire program to assume a lack of ambient authority. Most of existing code was written under no such assumption; effectively, existing applications need to undergo a rewrite-the-world migration to reap the advantages of ocap. We propose *gradual compartmentalization*, an approach which allows gradually migrating an application to object capabilities, component by component in arbitrary order, all the while continuously enjoying security guarantees. The approach relies on runtime authority enforcement and tracking the authority of objects the type system. We present Gradient, a proof-of-concept gradual compartmentalization extension to Scala which uses Enclosures and Capture Tracking as its key components. We evaluate our proposal by migrating the standard XML library of Scala to Gradient.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Theory of computation** → **Type structures**; • **Software and its engineering** → *Object oriented languages*; *Classes and objects.*

Additional Key Words and Phrases: type systems, security, object capabilities, compartmentalization

ACM Reference Format:
Aleksander Boruch-Gruszecki, Adrien Ghosn, Mathias Payer, and Clément Pit-Claudel. 2024. Gradient: Gradual Compartmentalization via Object Capabilities Tracked in Types. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 311 (October 2024), 27 pages. https://doi.org/10.1145/3689751

1 Introduction

Modern software development favors productivity over security. Application developers rely on diverse, unverified libraries written by unknown authors and downloaded off the Internet in order to extend their applications with basic functionality. In the extreme, modern application development becomes merely "gluing libraries together". This situation gave rise to *supply chain attacks*, a very dangerous attack vector. Finding a bug in a popular library, compromising a genuine one (e.g., by stealing its author's credentials) or publishing obfuscated malicious code can potentially grant access to hundreds of thousands of devices [Nikiforakis et al. 2012].

Authors' Contact Information: Aleksander Boruch-Gruszecki, aleksander.boruch-gruszecki@mff.cuni.cz, Charles University, Czechia; Adrien Ghosn, ghosn.adrien@gmail.com, Azure Research, Microsoft, UK; Mathias Payer, mathias.payer@epfl.ch, EPFL, Switzerland; Clément Pit-Claudel, clement.pit-claudel@epfl.ch, EPFL, Switzerland.

ACM 2475-1421/2024/10-ART311
https://doi.org/10.1145/3689751

Modern software needs fine-grained compartmentalization, i.e., intra-process isolation. Ideally, application developers should be able to enforce the Principle of Least Authority (or Privilege) [Melicher et al. 2017; Saltzer 1974]: any software component's access to program and system resources should be limited to the minimum required for its correct operation.

Object capabilities are a particularly attractive approach to compartmentalization with a long history of research [Dennis and Van Horn 1966; Melicher 2020; Miller 2006; Morris 1973; Rees 1996]. The ocap discipline views all code in terms of objects and specifies that access to program and system resources is mediated via special objects: *capabilities*. Capabilities originate from the program's entrypoint; objects can only access a capability they received from another object, i.e., there is no ambient authority in the system. Packages are also viewed as objects, called *modules* [Melicher et al. 2017]. Since a module can only use capabilities it received from other objects, an application can control the authority of its components by controlling how capabilities are distributed.

Despite their clear advantages, ocap languages (e.g., E [Miller 2006], Newspeak [Bracha et al. 2010] or Wyvern [Melicher et al. 2017]) are not widely used in the industry. Arguably, this is precisely *because* they assume an application's code to have no ambient authority: existing applications were not written under such an assumption. If their developers want to reap the benefits of ocap, they face an extensive rewrite of their entire codebase, including the very libraries they introduced to the codebase to reduce their own labor.

We develop an approach for compartmentalizing an application which allows a *gradual* migration to object capabilities. Code at various levels of migration can coexist within a single application; this not only allows introducing the object capability discipline to the application component by component, but *also* allows extending an application with non-ocap components while still maintaining our desired security guarantees. We are inspired by the idea of dynamically-enforced types from the literature on gradual typing [Siek and Taha 2006; Wadler 2015; Wadler and Findler 2009], which allows values (equivalently, objects) to be dynamically typed, which allows using them for any operation at the cost of potential runtime errors. We apply the concept of dynamic enforcement specifically to the *authority* of objects and not their entire types.

The key problem we solve is that until recently, it was unclear how to integrate existing non-ocap code with ocap code in a single application and still allow its components to be compartmentalized. The object capability discipline assumes no part of the system has ambient authority, while existing non-ocap code was written under no such assumption and may access arbitrary program and system resources. As a schematic example, in currently existing code a Log4J logger can simply be instantiated as follows.

```
(new log4j.Logger()).info("msg")
```

In contrast, an ocap version of the Logger class would need to explicitly take the capabilities to access the filesystem, the network, and eval arbitrary code [Chowdhury et al. 2022] as arguments.

```
(new log4j.Logger(fs, net, eval)).info("msg")
```

Ocap and non-ocap code seem to be fundamentally at odds. We alluded that mediating between them seems to inherently require *dynamically enforcing* the authority of non-ocap code. Such enforcement must be done *efficiently* enough to make the approach feasible in practice.

Recently, *Enclosures* [Ghosn et al. 2021] were proposed as an approach to compartmentalizing untrusted code which provides security guarantees even for foreign binaries thanks to relying on hardware support. An Enclosure restricts what program and system resources can be accessed in a given lexical scope; its restriction is expressed in terms of packages (and the memory associated with them) and system calls. Our key insight is that we can understand system calls as though they were method calls to a capability captured by the surrounding code, in addition to understanding mutable objects as capabilities. Doing so allows understanding existing non-ocap code as ocap code

which was already initialized with some capabilities, *and* allows restricting the authority of such code at runtime with an Enclosure-like mechanism.

Furthermore, we extend the type system to verify that all foreign code had its authority restricted, either via dynamic checks or statically, with the type system itself. We do so via *Capture Tracking* [Boruch-Gruszecki et al. 2023], a recently-published approach which augments types with *capture sets*, describing what capabilities each object has captured and therefore its authority. A particular advantage of Capture Tracking is its low annotation burden. Tracking the authority of objects in their types adds an *intermediate step* when migrating an application's component to object capabilities: the type system can be used to statically restrict the component's authority, without refactoring the component to take its desired capabilities as arguments.

Our contributions are as follows:

- *Gradual compartmentalization*, a hybrid approach has the advantages of both dynamically-enforced and statically-verified compartmentalization and allows a gradual migration from one approach to the other.
- We discuss *Gradient*, a proof-of-concept gradual compartmentalization extension to the Scala language, in order to illustrate the key principles of our approach.
- We show the GradCC system to demonstrate Capture Tracking can be used to track authority of mutable objects even in presence of capture-unchecked terms.
- We validate that migrating existing Scala code to capture-checked, non-ocap Gradient code is practical by migrating Scala's standard XML library.

The rest of the paper is organized as follows. First we discuss additional background and motivation behind our approach (Section 2). Next, we present Gradient (Section 3). We then present the formal system (Section 4, Section 5) and finally we evaluate Gradient based on the experience of us migrating a real-world Scala library and the feasibility of implementing Gradient (Section 6).

2 Background and Motivation

We distinguish and contrast between two salient ways of approaching compartmentalization: dynamic enforcement and static verification.

Dynamically-enforced compartmentalization is widely adopted in the industry. Examples include website sandboxing (e.g., Chromium tab isolation), containerization (e.g., Docker), systemd sandboxing, Linux application sandboxing (e.g., Snap, Flatpak), and mobile app permissions.

Dynamic mechanisms often operate at a coarse granularity, such as memory pages and processes. Compartmentalizing an existing application with such an approach is often challenging, requires heavy refactoring and incurs runtime costs. For instance, compartmentalizing an untrusted library with a process-based approach requires re-designing the application to run the untrusted code in a separate process and incurs the overhead of process switching and inter-process communication.

An important benefit of such low-level mechanisms is allowing *heterogenous* software written in any language, delivered as source code or as binaries. For instance, enclosures still provide security guarantees even in the presence of calls to foreign code which may forge arbitrary pointers.

Still, dynamic mechanisms naturally lead to runtime errors. Determining what policies to implement with such an approach is a matter of costly trial and error, since most software does not specify what permissions it needs. Overly broad policies weaken security; overly tight policies may cause runtime errors and prohibit expected functionality. Tellingly, Linux distributions do not agree on the systemd sandboxing restrictions placed on various services [Sandboxdb 2023] and the Java Security Manager was deprecated partly due to the "practically insurmountable challenge" [Java 2021] of determining appropriate security policies.

Statically-verified compartmentalization can be significantly more ergonomic, especially if it is integrated with a programming language. Such an approach inherently can assume code to be *homogenous*. It can tightly integrate enforcement of security policies with existing language constructs and types, dealing with objects rather than memory pages and system calls and scoping the restrictions to code blocks rather than to entire libraries. Such a mechanism can also statically verify if an application's components obey their intended system access restrictions, without incurring a performance cost *and* providing feedback quickly and reliably. Such feedback enables rapid development of security-conscious software and improves its maintainability: after any change, including a dependency update, security policies can be statically verified.

Gradual compartmentalization is a hybrid approach which lets the users adopt the best possible isolation strategy for each library:

1. Ocap code uses *object capabilities* as the principled compartmentalization mechanism.
2. Ocap code can interoperate with non-ocap code by leveraging *Capture Tracking* in order to track the authority of objects in types.
3. When all else fails, an Enclosures-inspired *runtime component* can dynamically enforce capability access restrictions, and Capture Tracking ensures the runtime component is used.

3 Gradient

The three key elements of gradual compartmentalization are object capabilities, tracking capabilities in types, and runtime authority enforcement. We present and discuss them based on Gradient, a proof-of-concept extension to the Scala programming language.

3.1 Object Capabilities

In Scala, similarly to most programming languages, ambient authority allows accessing system functionality simply by importing and using the appropriate packages. The object capability discipline, however, dictates that system functionality can be accessed only by calling methods on *capability* objects, which must be passed to their use-sites. Hence, Gradient code is organized in class-like units called *modules*, which have *constructors* and so can take arguments. If the code in a module needs capabilities, the module can request such capabilities as constructor arguments. (In all our examples, the modules retain their constructor arguments as private fields.) For example, the following snippet shows an example Gradient program's entrypoint.

```
module Main(fs: Fs^, net: Net^):
  def main() =
    val logger = new Logger(fs)
    ... // do useful work
```

The program defines the `Main` module with a `main` method. The module's constructor takes two capability arguments: `fs` and `net`. They implement the `Fs` and `Net` interfaces, respectively, and are marked as capabilities by the hat `^` sign. The program starts by instantiating the `Main` module with the appropriate capabilities and calling its `main` method.

The `main` method itself begins by instantiating the Logger module, passing it the `fs` capability as an argument. The Logger module is defined as follows.

```
module Logger(fs: Fs^):
  def log(msg: String): Unit = ... // log the message
```

Organizing the code into ocap modules has some major benefits. First, it facilitates inspecting what system functionality may be accessed by an untrusted module. The ocap discipline limits the number of ways a module can gain *direct* access to a capability: it can receive it as an argument to a constructor or a method, receive it as a method's result, or read it out of mutable state. This is a basis for reasoning about access to resources [Melicher 2020]. For instance, to convince ourselves that `Logger` cannot access the network, we begin by checking that it does not receive a capability

to do so as a constructor argument. By further inspecting its API, we see it cannot receive this capability as a method argument either, and so it cannot access the network as desired.

Second, modules allow easily *attenuating* [Miller 2006] the authority gained through capabilities. Since a capability is just an object, we can create a wrapper capability around it and inspect every method call and its arguments to decide if it should be allowed; in a sense, doing so injects bespoke filters between a capability and its calling context. For instance, the following snippet schematically shows how `Main` can restrict `Logger`[1] so that it can only access files in the "/var/log" directory.

```
module Main(fs: Fs^, net: Net^):
  def main() =
    val wfs = new Fs {
      def open(path: Path): FileHandle =
        if path.isRootedIn("/var/log") then fs.open(path)
        else throw IllegalArgumentException() }
    val logger = new Logger(wfs)
    ... // do useful work
```

The ocap discipline naturally allows compartmentalizing programs: security policies can be expressed by controlling the capabilities received by a module and attenuating their authority. E.g., if Log4j [Chowdhury et al. 2022; Hiesgen et al. 2022] was an ocap module, all programs using it would know it may access the network and load arbitrary code, since the Log4j module would request, say, the `net` and `eval` capabilities[2]; any module (transitively) using Log4j would either request the same capabilities or an initialized Log4j module instance. If Log4j only requested the `eval` module after a (perhaps malicious) update, programs upgrading to the new version would only compile after their code was intentionally modified to grant Log4j more authority. Finally, the Main module would naturally be able to attenuate the authority of capabilities passed to Log4j.

One puzzle piece remains: mutable state. If two modules share mutable state, they can communicate and defeat compartmentalization by exchanging either capabilities or behaviour-influencing information, leading to issues such as a confused deputy attack [Hardy 1988]. Ocap code is inherently more resistant to such attacks [Rajani et al. 2016]. Since it has no global mutable variables, two ocap modules can only communicate through mutable state if they received references (indirectly) pointing to the same mutable objects. Gradient goes further: it tracks all mutable objects in its type system, facilitating seeing if two modules can communicate. To explain this, next we present our approach to tracking capabilities in types: Capture Tracking.

3.2 Capture Tracking

Under Capture Tracking, every type has the form `S^{`$c_1, \ldots, c_n$`}`, e.g., `Logger^{fs}`. The *shape type* `S` describes the available operations, like normal types in most type system. The *capture set* `{`$c_1, \ldots, c_n$`}` describes the captured capabilities. We call types with non-empty capture sets *capturing types* and types with empty capture sets *pure types*. The latter can be written simply as `S` for brevity, e.g., `Int` instead of `Int^{}`. Function types are written as `(x: T1) -> T2`; as syntax sugar, a capturing function type can be written as `(x: T1) ->{`$c_1, \ldots, c_2$`} T2`. The result of a function type may mention the input parameter, making function types *dependent*.

Capabilities are naturally organized into a partial hierarchy: capturing a capability makes an object a derived capability, e.g., an instance of `Logger` which retains `fs` is a capability derived from `fs`. We posit that *all* capabilities derive their authority from other capabilities, and that ultimately all capabilities originate from **cap**, the *root capability*. **cap** is a static type system fiction; it exists to make the capability hierarchy a tree rooted in **cap**. As a shorthand, `S^` is the same as `S^{cap}`.

[1] Interestingly, here `Logger` itself is a capability that attenuates filesystem access granted by `fs`.

[2] Note: dynamically loaded *ocap modules* cannot access any resources unless explicitly given a capability (see Miller [2006]).

The capability hierarchy gives rise to a subtyping-like *subcapturing* relation, which combines subset inclusion and capability derivation; this relation is integrated into subtyping between capturing types. We illustrate this with the following example.

```
def configure(lgr: Logger^{cap}): Unit =
  ... // configure the logger

def mk_logger(fs: Fs^{cap}): Logger^{fs} =
  val lgr: Logger^{fs} = new Logger(fs)
  configure(lgr)
  return lgr
```

The `mk_logger` function builds a preconfigured `Logger`. It constructs `lgr`, an instance of `Logger` which retains `fs`, which makes it a capability derived from `fs` and gives it `{fs}` as its capture set. Next, `mk_logger` calls `configure` with `lgr`. Type-checking the call requires subtyping between `Logger^{fs}`, the type of `lgr`, and `Logger^{cap}`, the type of the formal parameter of `configure`. Capturing type subtyping requires capture set subcapturing, in this case `{fs} <: {cap}`, which we have since `fs` is bound at capture set `{cap}` and hence, is derived from **`cap`**.

`mk_logger` also shows how Capture Tracking approaches capture polymorphism.

```
def mk_logger(fs: Fs^{cap}): Logger^{fs}
```

The result of `mk_logger` mentions `fs`. Intuitively, the returned object captures the argument to `fs`, making its capture set *dependent* on the exact capture of that particular argument. If we call `mk_logger` with the real `fs` passed as an argument to `Main`, the result will be a capability; if we call it with an untracked `fs` which discards all writes, the result will be an untracked object.

3.2.1 Tracking Capabilities. Capabilities (and their capture) are tracked in Gradient types, which facilitates checking if a module may gain access to a capability. Consider this variant of `Logger`.

```
module Logger(fs: Fs^):
  def log(lvl: Level, msg: String): Unit
```

To verify that an instance of `Logger` cannot gain access to a capability other than `fs`, we need to check it cannot receive it as a method argument. Normally, we would need to inspect the implementations of all the parameter types (i.e., `Level` in our example) and verify they cannot store a capability. Capture Tracking simplifies this: we can instead inspect their capture sets alone. In our example, both `lvl` and `msg` are untracked (their types have empty capture sets), meaning that `Logger` cannot receive a capability reference as an argument.

Gradient treats all mutable objects as capabilities, which means that the above signature also lets us know `lvl` is untracked and therefore deeply immutable: it is itself immutable and does not grant access to (has not transitively captured) any mutable object.

3.2.2 Borrow Safety. We may want to restrict a module from retaining a capability. Consider the following scenario: `OCR`, a module for Optical Character Recognition defines a method (`update_models`) for downloading an updated version of its internal models from the network. The same module defines `ocr_pdf`, a method which OCRs a file. If it retained `net` after the call to `update_models` it could exfiltrate private information from the files it OCRs.

```
module OCR():
  def update_models(net: Net^): Unit = ...
  def ocr_pdf(stream: InputStream^): String =
    ... // OCR the contents of the stream

module Main(fs: Fs^, net: Net^):
  def main() =
    val ocr = new OCR()
    ocr.update_models(net)
    ... // do useful work involving OCR
```

Gradient uses Capture Tracking to statically rule out such problems and ensure *borrow safety*: a capability can only be retrieved out of mutable state if it is derived from other, already available

capabilities. In the above example, `OCR` cannot store `net` in its own state and retrieve it later: `net` is derived from **`cap`** and `OCR` did not (in fact, cannot) receive **`cap`** during its instantiation.

Note that *some* capabilities can be read from mutable state. As an example, we can loop over all files in a directory and keep the current file in a mutable variable.

```
def foo(fs: Fs^) =
  val iter : Iterator[File^{fs}] = fs.children(...)
  var file : File^{fs} = null
  while iter.has_next():
    file = iter.next()
    ... // do useful work involving file
```

3.3 Runtime-Assisted Graduality

Existing code accesses system features via system calls, and we can think of system calls as though they were method calls to a particular capability. These capabilities are clearly the most basic capabilities available to the program: we will call them *devices*. Ocap code receives the devices it needs as arguments to the `Main` module. On the other hand, non-ocap *packages* can be treated as though they were ocap modules which were *pre-instantiated* with the devices they may access. In addition, a package may optionally be *capture-unchecked*, which allows using any existing Scala package from Gradient code with no migration cost, but at the cost of relying on a runtime component to ensure the package does not exceed its authority.

When migrating a preexisting Scala package to ocap, the first step is to make it capture-checked. Since the code in the package can still be written as though it had ambient authority, this first step in many cases should only require adding capture signatures to existing code, without needing to refactor it (see Section 6). The package can later be refactored into a module by rewriting the code so that it accepts the devices it needs as arguments from code outside the module; doing so allows the users of the module to attenuate the authority they grant to the module, as described in Section 3.1. Such an architecture resembles the "dependency injection" design pattern and arguably is a good software engineering practice [Miller 2006].

3.3.1 Capture-Checked Packages. The following example uses a capture-checked `Logger` *package*.

```
module Main(device fs, device net, package Logger):
  def main() =
    Logger.log("Starting...")
    ... // do useful work and log it
```

The `Main` module is different compared to Section 3.1. It specifically requires the `fs` and `net` *devices*, as opposed to arbitrary objects implementing the `Fs` and `Net` interfaces. Since it uses the `Logger` *package*, it needs to explicitly request it as well. This does not mean we compromise ocap: `Logger` was pre-instantiated with devices before `Main` was instantiated. Since there's only one possible instance of `fs`, `net` and `Logger`, `Main` can request them by name without specifying their type, similarly to how import statements do not require a type. `Logger` itself is defined as follows.

```
package Logger:
  def log(msg: String): Unit = fs.open(...).write(msg)
```

In the source, `Logger.log` can directly access the `fs` device, not unlike existing Scala packages. However, Gradient interprets this definition differently from baseline Scala: the **`package`** statement logically defines a first-class object available in global lexical scope. Non-ocap code in other packages can directly use it, while ocap modules need to explicitly require it as an argument.

Packages on their own give weaker security guarantees than modules. Since `Logger` was already instantiated with devices, `Main` cannot compartmentalize it as before. The ability to attenuate

device access is lost; Capture Tracking, however, still tracks the devices captured by `Logger` in its type. Gradient allows using this information to check at compile time what devices `Logger` may access, using a **`restricted`** block.

```
module Main(device fs, device net, package Logger):
  def main() =
    restricted[{fs}] { Logger.log("Starting...") }
    ... // do useful work and log it
```

A **`restricted`** block is the Capture Tracking equivalent of a type ascription: the ascribed block (`{ Logger.log("Starting...") }` in the example) can only use capabilities from the ascription (`{fs}` in the example). For convenience, these ascriptions can be collected into a module.

```
module SafeLogger(device fs, package Logger):
  def log(msg: String): Unit =
    restricted[{fs}] { Logger.log(msg) }
```

In either case, thanks to Capture Tracking we re-gain the desired security guarantees. Our program statically checks if `Logger` accesses only the devices we permit it to access.

We have seen that we can integrate non-ocap and ocap code without compromising the compartmentalization guarantees or essential aspects of object capabilities. Still, we have assumed that the `Logger` package has capture signatures, i.e., it is capture-checked. Naturally, this is not the case for arbitrary existing Scala code: assigning it capture signatures involves a degree of manual work and in edge cases may require refactoring the code.

3.3.2 Capture-Unchecked Packages. As the name suggests, the signatures of a capture-unchecked package do not mention capture sets. This means we cannot rely on the **`restricted`** block to restrict the authority of such a package. Instead, the **`enclosed`** block can be used to dynamically restrict access to devices using a *runtime component*. The type system ensures that all code from capture-unchecked packages is run in an **`enclosed`** block. Syntactically, using a capture-unchecked package is similar to previous examples.

```
#package Logger:
  def log(msg: String): Unit = fs.open(...).write(msg)

module Main(device fs, device net, #package Logger):
  def main() =
    enclosed[{fs}] { Logger.log("Starting...") }
    ... // do useful work and log it
```

An **`enclosed`** block operates at a lower granularity than a **`restricted`** block: its restriction can only mention devices and *regions*. All modules have an associated memory region, and all mutable objects are at creation associated with such a region. The runtime component can efficiently check that only the specified devices and regions are accessed. If an **`enclosed`** block exceeds its restriction due to capture-unchecked code, its execution will be aborted and an exception will occur; capture-checked code can still be verified statically.

4 Base Formalism

The next two sections present the formal foundations for Gradient, split into two fragments. We start by presenting ModCC, the fragment which accounts for capture-checked Gradient programs and their features: modules, packages and mutable state. Later we present GradCC, which allows formally representing capture-unchecked code.

We use the notation $\overline{E_i}^i$ to denote a *syntactic repetition* of a non-negative number of syntax forms E_i. If the individual forms never occur alone, we omit the index as in $\overline{E}$. Furthermore, we write $\overline{E}^{0..1}$ to denote an *optional* occurence of E.

4.1 Syntax

We begin by giving an overview of ModCC and its syntax; we discuss how ModCC can formally represent Gradient programs at the end of this subsection.

ModCC is based on $\text{CC}_{<:\square}$ [Boruch-Gruszecki et al. 2023]. In short, $\text{CC}_{<:\square}$ is a capture-tracked version of System $\text{F}_{<:}$ with boxes and ANF-like terms, while ModCC is a type-monomorphic version of $\text{CC}_{<:\square}$ with modules, mutable state, and paths instead of variables. New forms and names of new rules are highlighted in gray.

A-normal form. Similarly to ANF, $\text{CC}_{<:\square}$ forces operands to be variables: $x\ y$ is a syntactically valid term and $(\lambda(x : T)\ t)\ y$ is not. This is not a loss of expressiveness. For instance, the general term-term application $t\ u$ can be expressed as $\textbf{let}\, x\ =\ t\, \textbf{in let}\, y\ =\ u\, \textbf{in}\, x\ y$. This approach has advantages in combination with dependent types: we effectively assign a name to every object. Programs may be written in the usual direct style, which can be elaborated by the compiler during type-checking as necessary, as is done by the Scala compiler.

Paths. ModCC allows selecting module fields with *paths* p. A path $x.\overline{f}$ is a *root variable* x followed by a zero or more *field selections* $.\overline{f}$. ModCC paths effectively replace $\text{CC}_{<:\square}$ variables: operands in terms and capture set elements both are paths, where they were variables in $\text{CC}_{<:\square}$. Additionally, ModCC paths can be looked up in a typing context $\Gamma(p) \longrightarrow T$ (**??**), much like classical variables can be looked up $x : T \in \Gamma$ (which we spell as $\Gamma \ni x : T$).

Dependent types. ModCC types may refer to a term variable if it occurs in a capture set; accordingly, function types have the form $\forall(x : T_1)\ T_2$ to allow x to occur in T_2.

Capture Tracking. ModCC types are partitioned into *shape types* S and regular types T. Syntactically, the latter are *capturing types* $S\,\hat{}\,C$, where the *capture set* C is a set of paths. We freely use shape types as regular types, assuming that $S \equiv S\,\hat{}\,\{\}$. Shape types comprise boxed types and the usual types of values. We refer to types with an empty capture set (incl. shape types) as *pure* types.

Boxes. We inherit box $\square\ p$ and unbox $C \multimap p$ forms from $\text{CC}_{<:\square}$. Boxing a capability *temporarily* prevents it from counting as captured by the surrounding term; its type also becomes a pure *boxed type* $\square\ T$. In order to use such a capability, first it needs to be *unboxed* $C \multimap p$, which can only be done at the cost of counting the capabilities in C as captured by the surrounding term.

Both box and unbox operations are statically inferred by the compiler [Boruch-Gruszecki et al. 2023]; they are specific to the formal system and not a feature of the surface language.

Boxes allow formally representing the interaction between capabilities and mutable state (Section 3.2.2): since the contents of mutable references Ref S must be pure, a capability can only be stored if it is boxed. An object that reads a capability out of mutable state has to unbox it before it can be used, which can only be done if the object's capture set accounts for the obtained capabilities.

References and regions. ModCC features *mutable references* which can be written to and read from. Each reference is associated with a *region* at creation. A reference is created with the $p.\textbf{ref}\ q$ form, where q is the initial value of the reference and p is a *region capability*, itself created with the **region** form. Importantly, no capability is necessary to create a region, which allows creating regions and using them to allocate local mutable state even inside untracked (pure) functions. References can be read from with the $!p$ form and written to with the assignment form $p := q$.

Records and modules. ModCC extends $\text{CC}_{<:\square}$ with records $\{\overline{f = p}\}$ and their usual semantics. In addition, ModCC also features *modules*: special records which can be created with the $\textbf{mod}(p)\ \{\overline{f = q}\}$ form. Doing so creates a record that packs together a region capability p with

Variable	x, y, z, **cap**			**Field**	f, **reg**
Path	p, q	::=	$x.\overline{f}$		
Value	v, w	::=	$\Box\, p \mid \lambda(x : T)\, t \mid \{\overline{f = p}\} \mid ()$		
Term	t, u	::=	$p \mid v \mid p\, p \mid C \multimap p \mid$ **let** $x = t$ **in** $t \mid$ **region**		
			$\mid\ !p \mid p := p \mid p.\mathbf{ref}\, p \mid \mathbf{mod}(p)\, \{\overline{f = p}\}$		
Shape Type	S, R	::=	$\top \mid \forall(x : T)\, T \mid \Box\, T \mid$ Unit		
			$\mid$ Ref $S \mid$ Reg $\mid\ \overline{\mu x}^{0..1}\, \{\overline{f : T}\}$		
Type	T, U	::=	$S \wedge C$		
Capture Set	C, D	::=	$\{\overline{p}\}$		
Typing Context	Γ, Δ	::=	$\emptyset \mid \Gamma, x : T$ **if** $x \neq$ **cap**		

Context lookup $\quad \Gamma \vdash p$ **bd** $\iff \exists T.\, \Gamma(p) \to T$ $\qquad \boxed{\Gamma(p) \to T}$

$$\frac{\Gamma \ni x : S \wedge C}{\Gamma(x) \to S \wedge C} \qquad \frac{\Gamma(p) \to \overline{\mu y}^{y}\, \{\overline{f_i : T_{f_i}}^{i}\} \wedge C}{\Gamma(p.f) \to [\overline{y := x}^{y}]T_f}$$

Fig. 1. ModCC syntax, context lookup. Highlighted forms are new compared to $\mathrm{CC}_{<:\Box}$.

other values $\overline{q}$ (the bodies of fields $\overline{f}$); the region capability is stored in the special field **reg** and the fields' bodies may reference the packed region capability. Both records and modules are typed with the record type $\overline{\mu x}\, \{\overline{f : T}\}$, which allows an optional recursive quantifier. In a sense, ModCC modules are like a specialized version of ML modules [Mitchell and Harper 1988]: a Gradient module is always parameterized with a single region. We borrow the idea of modeling objects as records from DOT [Amin et al. 2016]; our record type features a recursive qualifier analogous to variable-recursive types from DOT. The qualifier is useful specifically for modules whose fields reference the region packed together with the module.

Captured capabilities. In Section 3, we saw that capture sets allow reasoning about capabilities captured by Gradient objects. In ModCC, capture sets may contain *paths*, which statically are rooted in free variables. Such variables name objects and during evaluation will be substituted with store locations, which may contain capabilities. The basis used by ModCC typing to assign capture sets to values is the cv function, which calculates *captured paths* of terms. We define the cv function (Figure 2) to calculate such *captured paths*. Essentially, $\mathrm{cv}(t)$ is almost exactly the free variables of t, except that it accounts for boxing and ANF.

- A boxed path $\Box\, p$ does not count as captured. Dually, for an unbox form $C \multimap p$ only the "key" C counts as captured.
- A let-bound value, the v in **let** $x = v$ **in** t, is only considered captured if it, or paths rooted in its name x, are captured by t.

Using paths instead of variables (i.e., defining $\mathrm{cv}(x.f) \triangleq \{x.f\}$) increases the precision of cv when dealing with records and modules. The following example shows how this affects ModCC typing.

$$\mathsf{fn} : \ \forall(x : \{f_1 : \mathrm{Proc} \wedge \{\mathbf{cap}\}, f_2 : \mathrm{Proc} \wedge \{\mathbf{cap}\}\} \wedge \{\mathbf{cap}\})\ \ \{f_0 : \mathrm{Proc} \wedge \{x.f_1\}\} \wedge \{x.f_1\}$$
$$\mathsf{fn} = \lambda(x : \{f_1 : \mathrm{Proc} \wedge \{\mathbf{cap}\}, f_2 : \mathrm{Proc} \wedge \{\mathbf{cap}\}\} \wedge \{\mathbf{cap}\})\ \ \{f_0 = \lambda(x : \mathrm{Unit})\ x.f_1\ ()\}$$

The result of fn captures only $\{x.f_1\}$, i.e., a single field of x. As a consequence, if fn is called with an argument whose field f_1 is pure, the result of the call will be pure no matter what is captured by the other fields of the argument.

Definition 4.1 (Capture Set Operators). We define the following path-aware set operators.

$$C \ominus x \triangleq \{\, y.\overline{f} \in C \mid y \neq x \,\} \qquad x \propto C \triangleq x \in \{\, y \mid y.\overline{f} \in C \,\}$$

Definition 4.2 (Captured Paths). The captured paths of a term t are given by $\mathrm{cv}(t)$, defined as follows.

$$\begin{array}{lcll}
\mathrm{cv}(p) & \triangleq & \{p\} & \\
\mathrm{cv}(\square\, p) & \triangleq & \{\} & \\
\mathrm{cv}(\lambda(x : U)\, t) & \triangleq & \mathrm{cv}(t) \ominus x & \\
\mathrm{cv}(\{\overline{f_i = p_i}^{\,i}\}) & \triangleq & \{\overline{p_i}^{\,i}\} & \\
\mathrm{cv}(\mathbf{let}\, x = v\, \mathbf{in}\, t) & \triangleq & \mathrm{cv}(t) & \mathbf{if}\ x \not\propto \mathrm{cv}(t) \\
\mathrm{cv}(\mathbf{let}\, x = u\, \mathbf{in}\, t) & \triangleq & \mathrm{cv}(u) \cup \mathrm{cv}(t) \ominus x & \\
\mathrm{cv}(p\, q) & \triangleq & \{p, q\} & \\
\mathrm{cv}(C \multimap p) & \triangleq & C \cup \{p\} & \\
\mathrm{cv}(p.\mathbf{ref}\, q) & \triangleq & \{p, q\} & \\
\mathrm{cv}(\mathbf{mod}(p)\, \{\overline{f_i = q_i}^{\,i}\}) & \triangleq & \{p, \overline{q_i}^{\,i}\} & \\
\mathrm{cv}(\mathbf{region}) & \triangleq & \{\} &
\end{array}$$

Fig. 2. The definitions of capture set operators and cv.

Gradient and ModCC. Gradient module definitions correspond to a formal ModCC term much like Scala class definitions correspond to a DOT term [Amin et al. 2016; Martres 2023]. Concretely, a module definition corresponds to a Gradient function which formally represents the module's constructor: it takes the constructor's arguments, creates a fresh region for the module and creates the module itself, as illustrated in the following example.

```
module Logger(fs: Fs^):
  def log(msg: String): Unit = ...
```

```
let newLogger = λ(fs: Fs^)
  let r = region in
  let _log = λ(msg: String) = ... in
  mod(r) { log = _log }
in
```

```
module Main(fs: Fs^, net: Net^):
  def main() =
    val logger = new Logger(fs)
    ...
```

```
let newMain = λ(fs: Fs^) λ(net: Net^)
  let r = region in
  let _main = λ(u: Unit)
    let logger = newLogger fs in ...
  in mod(r) { main = _main }
in
// initialize Main & run the program
let main = newMain fs net in
main.main ()
```

The example also shows that a Gradient program corresponds to a ModCC *term*. The module and package definitions the Gradient program comprises all correspond to let-bound ModCC terms; the body of the innermost let term initializes the packages (if there are any) and the `Main` module, and proceeds to run the program by calling the `main` method. We treat Gradient devices such as `fs` and `net` (and their types) as extensions to the base formalism; we do not privilege any particular device by baking it into the formal system.

4.2 Subcapturing

Figure 3 shows the subcapturing rules of ModCC. Subcapturing consistently uses paths instead of variables; accordingly, rule (SC-PATH) uses path lookup and replaces rule (SC-VAR) from $\mathrm{CC}_{<:\square}$,

Subcapturing $\boxed{\Gamma \vdash C <: C}$

SC-PATH
$$\frac{\Gamma(p) \to S \,\hat{}\, C}{\Gamma \vdash \{p\} <: C}$$

SC-ELEM
$$\frac{p \in D}{\Gamma \vdash \{p\} <: D}$$

SC-MEM
$$\frac{\Gamma \vdash p.f\ \mathbf{bd}}{\Gamma \vdash \{p.f\} <: \{p\}}$$

SC-SET
$$\frac{\overline{\Gamma \vdash \{p_i\} <: D}^{\,i}}{\Gamma \vdash \{\overline{p_i}^{\,i}\} <: D}$$

SC-TRANS
$$\frac{\Gamma \vdash C_1 <: C_2 \quad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash C_1 <: C_3}$$

Subtyping $\boxed{\Gamma \vdash T <: T}$

CAPT
$$\frac{\Gamma \vdash S_1 <: S_2 \qquad \Gamma \vdash C_1 <: C_2}{\Gamma \vdash S_1 \,\hat{}\, C_1 <: S_2 \,\hat{}\, C_2}$$

TOP
$$\Gamma \vdash S <: \top$$

REFL
$$\Gamma \vdash T <: T$$

TRANS
$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

BOXED
$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \square\, T_1 <: \square\, T_2}$$

FUN
$$\frac{\Gamma \vdash U_2 <: U_1 \quad \Gamma, x : U_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : U_1)\, T_1 <: \forall(x : U_2)\, T_2}$$

REC
$$\frac{\overline{\Gamma \vdash U_{f_j} <: T_{f_j}}^{\,j}}{\Gamma \vdash \{\overline{f_i : U_{f_i}}^{\,i}\} <: \{\overline{f_j : T_{f_j}}^{\,j}\}}$$

Typing $\boxed{\Gamma \vdash t : T}$

UNIT
$$\Gamma \vdash () : \mathsf{Unit}$$

PATH
$$\frac{\Gamma(p) \to S \,\hat{}\, C}{\Gamma \vdash p : S \,\hat{}\, \{p\}}$$

UNPACK
$$\frac{\Gamma \vdash p : \mu x\, \{\overline{f : T}\} \,\hat{}\, C}{\Gamma \vdash p : ([x := p]\{\overline{f : T}\}) \,\hat{}\, C}$$

PACK
$$\frac{\Gamma \vdash p : ([x := p]\{\overline{f : T}\}) \,\hat{}\, C}{\Gamma \vdash p : \mu x\, \{\overline{f : T}\} \,\hat{}\, C}$$

ABS
$$\frac{\Gamma, x : U \vdash t : T \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash \lambda(x : U)\, t : (\forall(x : U)\, T) \,\hat{}\, (\mathrm{cv}(t) \ominus x)}$$

APP
$$\frac{\Gamma \vdash p : \forall(x : U)\, T \,\hat{}\, C \qquad \Gamma \vdash q : U}{\Gamma \vdash p\, q : [z := q]T}$$

LET
$$\frac{\Gamma \vdash u : T \qquad \Gamma, x : T \vdash t : U \qquad x \notin \mathrm{fv}(U)}{\Gamma \vdash \mathbf{let}\, x = u\, \mathbf{in}\, t : U}$$

SUB
$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash t : U}$$

BOX
$$\frac{\Gamma \vdash p : S \,\hat{}\, C \qquad \overline{\Gamma \vdash q\ \mathbf{bd}}^{\,q \in C}}{\Gamma \vdash \square\, p : \square\, S \,\hat{}\, C}$$

UNBOX
$$\frac{\Gamma \vdash p : \square\, S \,\hat{}\, C \qquad \overline{\Gamma \vdash q\ \mathbf{bd}}^{\,q \in C}}{\Gamma \vdash C \multimap p : S \,\hat{}\, C}$$

REGION
$$\Gamma \vdash \mathbf{region} : \mathrm{Reg} \,\hat{}\, \{\mathbf{cap}\}$$

REF
$$\frac{\Gamma \vdash p : \mathrm{Reg} \,\hat{}\, \{\mathbf{cap}\} \qquad \Gamma \vdash q : S}{\Gamma \vdash p.\mathbf{ref}\, q : (\mathrm{Ref}\, S) \,\hat{}\, \{p\}}$$

READ
$$\frac{\Gamma \vdash p : (\mathrm{Ref}\, S) \,\hat{}\, \{\mathbf{cap}\}}{\Gamma \vdash\, !p : S}$$

WRITE
$$\frac{\Gamma \vdash p : (\mathrm{Ref}\, S) \,\hat{}\, \{\mathbf{cap}\} \qquad \Gamma \vdash q : S}{\Gamma \vdash p := q : \mathsf{Unit}}$$

RECORD
$$\frac{\overline{\Gamma \vdash p_i : S_i \,\hat{}\, C_i}^{\,i}}{\Gamma \vdash \{\overline{f_i = p_i}^{\,i}\} : \{\overline{f_i : S_i \,\hat{}\, C_i}^{\,i}\} \,\hat{}\, (\bigcup_i C_i)}$$

MODULE
$$\frac{\Gamma \vdash q : \mathrm{Reg} \,\hat{}\, \{\mathbf{cap}\} \qquad \overline{\Gamma \vdash p_i : U_i}^{\,i} \qquad \overline{T_i = [q := x.\mathbf{reg}]U_i}^{\,i}}{\Gamma \vdash \mathbf{mod}(q)\, \{\overline{f_i = p_i}^{\,i}\} : \mu x\, \{\mathbf{reg} : \mathrm{Reg} \,\hat{}\, \{\mathbf{cap}\}, \overline{f_i : T_i}^{\,i}\} \,\hat{}\, \{\mathbf{cap}\}}$$

Fig. 3. ModCC static rules. Highlighted rules and premises are new or changed (resp.) compared to $\mathrm{CC}_{<:\square}$.

which only looked up variables. Rules (SC-ELEM) and (SC-SET) are directly inherited from $\mathrm{CC}_{<:\square}$. In addition, rule (SC-MEM) allows relating a module's field to the module itself. For simplicity, ModCC subcapturing features a separate transitivity rule (SC-TRANS), while transitivity in $\mathrm{CC}_{<:\square}$ was inlined into premises of subcapturing rules and therefore an admissible property.

4.3 Subtyping

Nearly all subtyping rules of ModCC are inherited from $CC_{<:\square}$ (Figure 3). Like in $CC_{<:\square}$, rule (CAPT) connects subtyping to subcapturing. Rule (REC) is the standard breadth-and-width rule for subtyping records. Since reference types are invariant, they do not have a dedicated subtyping rule and can only be compared with (REFL). Like in the DOT family of systems, recursive types do not participate in subtyping; instead, they can be eliminated and introduced in typing.

4.4 Typing

Figure 3 presents our typing rules. We inherit all the typing rules of $CC_{<:\square}$, with small adjustments to account for paths replacing variables: the (VAR) variable typing rule from $CC_{<:\square}$ is replaced with the path typing rule (PATH), while the $CC_{<:\square}$ rules (BOX) and (UNBOX) use path lookup to ensure that the typing context binds all capture set elements. Rules (PACK) and (UNPACK) allow packing and unpacking recursive qualifiers on module types. Rule (ABS) types term abstractions; it uses the cv of the abstraction term as the assigned capture set. Rule (APP) types term applications $p\,q$. Since the result of a function type may depend on its parameter, (APP) replaces such parameter occurences with the concrete argument applied to the abstraction. Rules (LET), (SUB) are standard.

Rules (REGION) and (REF) type region and reference creation forms, respectively. The capture set assigned to a reference is the region capability used to create it. Rule (REF) ensures that only pure, untracked objects can be stored in references. As explained in Section 4.1, this forces tracked objects to be boxed before they can stored in mutable state. If a tracked object is read out of mutable state, it needs to be unboxed before it can be accessed. Doing so adds the capabilities used to unbox the object to the cv of the unboxing term, which guarantees borrow safety (Section 3.2.2).

Rules (READ) and (WRITE) type read and write forms. Finally, rule (RECORD) is the standard record typing rule; the capture set of a record is the union of the field capture sets. Rule (MODULE) is a variant of (RECORD): a ModCC module is a record "packed" together with a region. For each field f_i, the rule requires the field's body p_i, to be typeable at some type U_i. However, in the entire module's type the type of each field f_i is instead $[q := x.\mathbf{reg}]U_i$, where q is the region packed with the module and x is the DOT-like *recursive self-reference*. Finally, since a module packs a region into itself, the capture set of the entire module is simply $\{\mathbf{cap}\}$.

4.5 Reduction

Figure 4 shows our reduction rules and runtime-specific forms. Unlike $CC_{<:\square}$, ModCC reduces store-term *configuration* pairs (σ, t). The term t is decomposed into an *evaluation context* η and a potential redex u. The rules are deterministic: at any point there is at most one applicable rule.

Reduction rules use l for store locations and r for paths rooted in locations. Rather than treating locations them as a different grammatical category and defining additional typing rules, we make the simplifying assumption that they are variables. Stores σ comprise location-entry pairs $l \mapsto e$; an entry e is either a value, a region, a region-associated reference or a module.

Rules (APPLY), (TAPPLY), (OPEN), (RENAME) and (LIFT) are inherited from $CC_{<:\square}$. Rules (GET) and (SET) reduce mutable state reads and writes. Reference and module creation forms are reduced by rules (ALLOC) and (MALLOC). Because the fields of records and modules in the store always point to other paths and ultimately resolve to a location, runtime paths are effectively aliases for locations. Store lookup is aware of such aliases, e.g., given $\sigma = l_1 \mapsto \{f = l_2\}, l_2 \mapsto v$ we have $\sigma(l_1.f) = \sigma(l_2) = v$.

4.6 Metatheory

We show that ModCC is sound with the standard Progress and Preservation theorems [Wright and Felleisen 1994]. In our metatheory, we follow the Barendregt convention and only consider typing contexts where all variables are unique: for all contexts of the form $\Gamma, x : T$ we have $x \notin \mathrm{dom}(\Gamma)$.

As usual, we need to define store typing $\sigma \sim \Delta$. (As a convention, we use Δ to refer to typing contexts related to stores.) We define it in terms of store entry typing $\Delta \vdash l \mapsto e \sim \Delta$ as follows:

Definition 4.3. We have $\overline{l_i \mapsto e_i}^{\,i} \sim \Delta$ if and only if:

1. We have both $\overline{\Delta \vdash l_i \mapsto e_i \sim \Delta_i}^{\,i}$ and $\Delta = \overline{\Delta_i}^{\,i}$.
2. If e_i is a record $\{\overline{f_j = r_j}^{\,j}\}$ or a module $\mathbf{mod}(r')\ \{\overline{f_j = r_j}^{\,j}\}$, then for some T_i we have $\Delta = \Delta', l_i : T_i, \Delta''$ and we have $\overline{\Delta' \vdash r_j\ \mathbf{bd}}^{\,j}$.

The first condition connects store typing to store entry typing: the typing context of the former must be assembled out of fragments built by the latter. The second condition is a well-formedness criterion for stores: bodies of modules can only refer to paths bound *before* the module is bound. and likewise for records. Most of the store entry typing rules are the same as their corresponding typing rules ((UNIT), (BOX), (ABS), (REF)), e.g., if $\Delta \triangleq l_1 : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}, l_2 : (\mathrm{Ref}\ \mathrm{Unit})\,\hat{}\,\{\mathbf{cap}\}$ we have both $\Delta \vdash l_1 \mapsto \mathbf{region}_{l_1} : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}$ and $\Delta \vdash l_2 \mapsto l_1 \triangleright \mathbf{ref}\ () \sim l_2 : \mathrm{Ref}\ \mathrm{Unit}\,\hat{}\,\{\mathbf{cap}\}$.

The store typing rules for records and modules add *path aliases* $p \equiv q$ to the output (Figure 5) compared to normal typing rules; the syntax of typing contexts is extended to allow such aliases. Path aliases are necessary because the identity of regions is important during reduction. A direct path to a region must be the same as a path going through the **reg** of a module. Furthermore, the path packed with a module may refer to a region indirectly, through a record or a module field; such paths must be treated the same as the paths in the corresponding field's body. All mutually aliased paths must be bound to an equivalent type in the typing context corresponding to the store; this is enforced by the path lookup premise in both (ST-RECORD) and (ST-MODULE). Path aliases can be seen as a minimalistic version of singleton types studied in pDOT [Rapoport and Lhoták 2019]. Using path lookup instead of typing resembles *strict typing* used in the soundness proofs of the

Reduction $\boxed{(\sigma; t) \longrightarrow (\sigma; t)}$

$$
\begin{array}{lll}
(\sigma; \eta[\ r\ r'\ ]) & \longrightarrow (\sigma\ ; \eta[\ [x := r']t\ ]) \quad \textbf{if}\ \sigma(r) = \lambda(x : T)\ t & (\textsc{apply}) \\
(\sigma; \eta[\ C \multimap r\ ]) & \longrightarrow (\sigma\ ; \eta[\ r'\ ]) \quad \textbf{if}\ \sigma(r) = \square\ r' & (\textsc{open}) \\
(\sigma; \eta[\ !r\ ]) & \longrightarrow (\sigma\ ; \eta[\ v\ ]) \quad \textbf{if}\ \sigma(r) = l \triangleright \mathbf{ref}\ v & (\textsc{get}) \\
(\sigma; \eta[\ r := r'\ ]) & \longrightarrow (\sigma'; \eta[\ ()\ ]) \quad \textbf{if}\ \sigma' = [r \mapsto \sigma(r')]\sigma & (\textsc{set}) \\
(\sigma; \eta[\ v\ ]) & \longrightarrow (\sigma, l \mapsto v; \eta[\ l\ ]) \quad \textbf{if}\ l\ \text{fresh} & (\textsc{lift}) \\
(\sigma; \eta[\ \mathbf{let}\ x = r\ \mathbf{in}\ t\ ]) & \longrightarrow (\sigma\ ; \eta[\ [x := r]t\ ]) & (\textsc{rename}) \\
(\sigma; \eta[\ r.\mathbf{ref}\ r'\ ]) & \longrightarrow (\sigma, l \mapsto e; \eta[\ l\ ]) & \\
 & \quad \textbf{if}\ l\ \text{fresh},\ \sigma(r) = \mathbf{region}_{l'},\ e = l' \triangleright \mathbf{ref}\ \sigma(r') & (\textsc{alloc}) \\
(\sigma; \eta[\ \mathbf{mod}(r)\ \{\overline{f = r'}\}\ ]) & \longrightarrow (\sigma, l \mapsto e; \eta[\ l\ ]) & \\
 & \quad \textbf{if}\ l\ \text{fresh},\ \sigma(r) = \mathbf{region}_{l'},\ e = \mathbf{mod}(l')\ \{\overline{f = r'}\} & (\textsc{malloc})
\end{array}
$$

Variable	$l, \ldots$		
Store context	σ	::=	$\overline{l \mapsto e}$
Eval context	η	::=	$[\,] \mid \mathbf{let}\ x = \eta\ \mathbf{in}\ t$
Store entry	e	::=	$v \mid \mathbf{region}_l \mid l \triangleright \mathbf{ref}\ v \mid \mathbf{mod}(l)\ \{\overline{f = r}\}$
Runtime path	r	::=	$l \mid r.f$

Fig. 4. ModCC operational semantics.

Store entry typing $\boxed{\Delta \vdash l \mapsto e \sim \Delta}$

$$\textsc{st-record}\quad \frac{\overline{\Delta(r_i) \to T_i}^{\,i}}{\Delta \vdash l \mapsto \{\overline{f_i = r_i}^{\,i}\} \sim l : \{\overline{f_i : T_i}^{\,i}\}, \overline{l.f_i \equiv r_i}^{\,i}}$$

$$\textsc{st-module}\quad \frac{\Delta \vdash l : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\} \qquad \overline{\Delta(r_i) \to U_i}^{\,i} \qquad \overline{T_i = [l := x.\mathbf{reg}]U_i}^{\,i}}{\Delta \vdash l_0 \mapsto \mathbf{mod}(l)\,\{\overline{f_i = r_i}^{\,i}\} \sim l_0 : \mu x\,\{\overline{f_i : T_i}^{\,i}\}, l \equiv l_0.\mathbf{reg}, \overline{l.f_i \equiv r_i}^{\,i}}$$

Fig. 5. Abridged store entry typing rules of ModCC. See the technical report for the full version.

```
#package Logger:
  def log(msg: String): Unit =
    fs.open(...).write(msg)
```

```
let newLogger = λ(fs: Fs^#)
  let r = region in
  let _log = λ(msg: String)
    let h = #fs.open (...) in
    #h.write msg
  in mod(#r) { log = #_log }
in ...
```

Fig. 6. Representing an example capture-unchecked package in GradCC (see Section 3.3.2).

DOT systems [Amin et al. 2014; Boruch-Gruszecki et al. 2022; Rapoport and Lhoták 2019]. Path aliasing, like strict typing, is a proof device rather than a core feature of ModCC. They are parts of the metatheory: an alternative proof without them would be equally valid.

We state Progress and Preservation as follows (see the technical report for the proofs).

Theorem 4.4 (Progress). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then either there exists r such that $t = r$, or there exist σ', t' such that $(\sigma, t) \longrightarrow (\sigma', t')$.*

Theorem 4.5 (Preservation). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then $(\sigma, t) \longrightarrow (\sigma', t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$.*

5 Formalising Capture-Unchecked Terms

We introduce GradCC, which extends ModCC to allow formally representing capture-unchecked code. It is inspired by *casts* found in gradual typing [Siek and Taha 2006; Wadler 2015; Wadler and Findler 2009]. Casts allow representing type-unchecked code with terms where the types of all expressions were erased to the dynamic type Dyn. An extra cast is necessary to use such expressions, e.g., to increment an expression's result, type-unchecked code first casts it to Int.

GradCC takes a similar approach: it adds a *mark* form $\#\,p$, which replaces a path's capture set with a *mark* $\#$, marking the path as capture-unchecked. Accordingly, GradCC capturing types $S\,\hat{}\,C?$ are equipped with a *capture descriptor* $C?$, which is either a capture set C as in ModCC, or a mark $\#$. Capture sets themselves still only contain unmarked paths, but they may be *improper* if they can be widened through subcapturing so that they contain a path whose capturing type features a mark, i.e., is of the form $S\,\hat{}\,\#$. By extension, a path p is improper iff the capture set $\{p\}$ is improper. Intuitively, an improper path allows (indirectly) accessing an actual capture-unchecked, marked path. Figure 6 shows how GradCC can formally model a capture-unchecked package.

Dynamic restrictions. A marked path can be used similarly to a capture-checked one, e.g., it can be called if it stands for a function. However, its type does not specify a capture set and we no longer know what capabilities may be accessed through the marked path, which also applies

to improper paths (a superset of marked paths) in general. Hence, GradCC features the *enclosed term form* **encl**$[C][T]\ t$, the Capture Tracking equivalent of a type assertion: it allows dynamically restricting what capabilities t may access. The capture set C is a *restriction*: it lists the capabilities which accessible by t. The intention behind this form, and Gradient **enclosed** blocks in general, is to support an efficient implementation (see Section 6.2). Hence, GradCC restrictions only allow regions. Gradient restrictions also allow devices, which we treat as an extension to base GradCC.

Obscuring marks. Capture-unchecked code may access capture-checked definitions. However, an improper path p cannot be directly passed to code expecting a proper path, e.g., in map list p (representing a call to `List#map`). Instead, an *obscur* form must be used to *temporarily* treat p as though its capture set was {**cap**}, as in **obscur** p **as** f **in** map list f. An **obscur** form can only be used in the extent of an **encl** form, i.e., with a dynamic authority restriction in place. Typing ensures f is only accessed in the dynamic extent of its lexical scope (Section 5.1).

Marks and boxes. Capture-unchecked code also needs to interact with boxes. First, improper paths are still tracked and need to be boxed before being written to mutable state. Second, as capabilities read out of capture-checked mutable references must be unboxed, improper paths can be boxed *and* boxes can be opened with a mark-open form $\# \multimap p$.

Definition 5.1 (Capture Descriptor Operators). We extend capture set operators to capture descriptors as follows. Note that # is effectively empty according to both $\dot{\in}$ and $\dot{\propto}$.

$$\begin{array}{lll} \# \dot{\cup} C? \triangleq \# & \# \dot{\ominus} x \triangleq \# & p \dot{\in} C \triangleq p \in C \\ C? \dot{\cup} \# \triangleq \# & C \dot{\ominus} x \triangleq \{\, y.\overline{f} \in C \mid y \neq x \,\} & p \dot{\propto} C \triangleq p \propto C \\ C_1 \dot{\cup} C_2 \triangleq C_1 \cup C_2 & & p \dot{\notin} \# \quad x \dot{\not\propto} \# \end{array}$$

Definition 5.2. We extend cv as follows. Previous rules use capture descriptor operators instead of capture set operators.

$$\begin{array}{lcl} \mathrm{cv}(\#\, p) & \triangleq & \# \\ \mathrm{cv}(\textbf{encl}[C'][S \hat{}\, C?]\ t) & \triangleq & C? \dot{\cup} C' \\ \mathrm{cv}(\textbf{obscur}\ p\ \textbf{as}\ x\ \textbf{in}\ t) & \triangleq & \# \\ \mathrm{cv}(\# \multimap p) & \triangleq & \# \end{array}$$

5.1 Changes to the System

Figure 7 shows the complete syntax of the new GradCC forms, and the new typing and subtyping rules. The rules use the following auxilliary definition.

Definition 5.3 (Well-formed Restriction). C is a well-formed restriction in Γ, or $\Gamma \vdash C$ **wfr**, iff we have $C = \{\overline{x_i}^i\}$ such that $\overline{\Gamma \vdash x_i : \mathrm{Reg} \hat{}\, D_i}^i$ for some $\overline{D_i}^i$.

GradCC subcapturing is the same as in ModCC. Subcapturing still relates capture sets C, not capture descriptors $C?$. Subtyping is extended with (MARKED), which relates marked types. While $S \hat{}\, C$ and $S \hat{}\, \#$ are unrelated via subtyping, a term can be converted from $S \hat{}\, C$ to $S \hat{}\, \#$ by marking it. Typing is extended with four rules, one per each new term form. Notably, (OBSCUR) introduces x as a *scoped* capability, ensuring it is not accessed after the **obscur** form is left. Only a boxed capability can leave another capability's scope, and we can only open a box if the capability inside (its capture set) is *accounted for* (subcaptures) currently bound capabilities. Boruch-Gruszecki et al. [2023] show that since scoped capabilities are not accounted for by any capability bound outside of their scope, this check ensures they can only be accessed within the dynamic extent of their scope. A GradCC capability can leave another capability's scope only by being returned from it or through mutable state; both channels only allow pure objects, forcing capabilities to be boxed.

Unmarked path	ρ	::=	$x.\overline{f}$
Stable path	p, q	::=	$\rho \mid \#\rho$
Term	t, u	::=	$\ldots \mid \mathbf{encl}[C][T]\,t \mid \mathbf{obscur}\,p\,\mathbf{as}\,x\,\mathbf{in}\,t \mid \# \multimap r$
Type	T, U	::=	$S\,\hat{}\,C?$
Capture descriptor	$C?$	::=	$C \mid \#$
Capture set	C	::=	$\{\,\overline{\rho}\,\}$

Subtyping and typing $\boxed{\Gamma \vdash T <: T}$ $\boxed{\Gamma \vdash t : T}$

MARKED
$$\frac{\Gamma \vdash S_1 <: S_2}{\Gamma \vdash S_1\,\hat{}\,\# <: S_2\,\hat{}\,\#}$$

ENCLOSURE
$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash C\ \mathbf{wfr}}{\Gamma \vdash \mathbf{encl}[C][T]\,t : T}$$

OBSCUR
$$\frac{\Gamma \vdash p : S\,\hat{}\,C? \qquad \Gamma, x : S\,\hat{}\,\{\mathbf{cap}\} \vdash t : R}{\Gamma \vdash \mathbf{obscur}\,p\,\mathbf{as}\,x\,\mathbf{in}\,t : R}$$

MARK
$$\frac{\Gamma \vdash p : S\,\hat{}\,C?}{\Gamma \vdash \#p : S\,\hat{}\,\#}$$

UNBOX-MARK
$$\frac{\Gamma \vdash p : \square\, S\,\hat{}\,C \qquad \overline{\Gamma \vdash q\ \mathbf{bd}}^{\,q \in C}}{\Gamma \vdash \# \multimap p : S\,\hat{}\,C}$$

Fig. 7. GradCC static extensions. Note capture sets (hence, types) only allow unmarked paths ρ.

5.2 Reduction

GradCC operational semantics are defined in terms of two reduction relations (Figure 8). The "underlying" relation $\cdot \longrightarrow \cdot$ relates two configurations and extends the ModCC reduction relation. The "primary" relation $\cdot \longrightarrow_e \cdot$ enforces runtime restrictions of enclosures. It specifies that a configuration reduces according to $\cdot \longrightarrow \cdot$ iff the redex is permitted in the current restriction; otherwise the configuration reduces to **fail**. The redexes for creating a reference, reading to it or writing to it are permitted only if the involved region is within the current restriction; other redexes are always permitted. These semantics match Enclosures [Ghosn et al. 2021], which stop the program if it accesses memory outside of the current restriction.

5.3 Metatheory

The statement of soundness for GradCC is a bit more involved compared to ModCC, since well-typed programs may contain capture-unchecked fragments and thus inherently can reduce to **fail**. This occurs when a capture-unchecked fragment violates the restriction of an **encl** form. Such results are expected, as in gradual typing systems; we do not want our correctness theorems to disallow them. Instead, we ensure that cv can be used to predict what capabilities may be accessed, as in the original $\mathrm{CC}_{<:\square}$ system [Boruch-Gruszecki et al. 2023].

To see why this is important, we need to take a step back: $\mathrm{cv}(t)$, used by typing to assign capture sets, describes the capabilities referenced by t. It is almost the free variables of t, except that it accounts for boxing[3] and **encl** forms. The latter allow dynamically limiting access only to capabilities in their *restriction*; their cv counts their restriction in lieu of inspecting their body. Hence, if we can use $\mathrm{cv}(t)$ to statically reason about capabilities accessed when reducing t, in particular the restrictions of **encl** forms are meaningful. Such reasoning also formally grounds using capture sets to reason about program and system resource access in Gradient programs, since in Gradient such access is always mediated via (device) capabilities.

[3]The cv function ignores boxed capabilities and only counts the "key" $C?$ of an unbox form $C? \multimap x$; hence, boxed capabilities cannot be used until unboxed and the "key" must account for an unbox form's result (see [Boruch-Gruszecki et al. 2023]).

Reduction $\boxed{(\sigma;t) \longrightarrow (\sigma;t)}$ $\boxed{(\sigma;t) \longrightarrow_e ((\sigma;t) \mid \textbf{fail})}$

Eval context $\eta ::= \ldots \mid \textbf{encl}[C][T]\,\eta$

$$
\begin{array}{lcll}
(\sigma;\eta[\ C?\ \circ\!\!-\ r\ ]) & \longrightarrow & (\sigma;\eta[\ r'\ ]) & \textbf{if } \sigma(r) = \square\, r' \quad (\textsc{open}) \\
(\sigma;\eta[\ \textbf{encl}[C][T]\ r\ ]) & \longrightarrow & (\sigma;\eta[\ r\ ]) & (\textsc{exit}) \\
(\sigma;\eta[\ \textbf{obscur}\, r\, \textbf{as}\, x\, \textbf{in}\, t\ ]) & \longrightarrow & (\sigma;\eta[\ [x := r]t\ ]) & (\textsc{obs})
\end{array}
$$

$$
\frac{(\sigma;\eta[\ t\ ]) \longrightarrow (\sigma';\eta'[\ t'\ ]) \qquad \mathcal{A}(\sigma,t) \subseteq \mathcal{R}(\sigma,\eta)}{(\sigma;\eta[\ t\ ]) \longrightarrow_e (\sigma';\eta'[\ t'\ ])}
\qquad
\frac{\mathcal{A}(\sigma,t) \not\subseteq \mathcal{R}(\sigma,\eta)}{(\sigma;\eta[\ t\ ]) \longrightarrow_e \textbf{fail}}
$$

$$
\mathcal{R}(\sigma, \textbf{encl}[\{\overline{r_i}^{\,i}\}][T]\,\eta) \triangleq \{\overline{l_i}^{\,i}\} \cap \mathcal{R}(\sigma,\eta) \qquad \textbf{if } \overline{\sigma(r_i) = \textbf{region}_{l_i}}^{\,i}
$$

$$
\mathcal{R}(\sigma, \textbf{let}\, x = \eta\, \textbf{in}\, t) \triangleq \mathcal{R}(\sigma,\eta) \qquad \mathcal{R}(\sigma,[\,]) \triangleq \{\}
$$

$$
\begin{array}{lcll}
\mathcal{A}(\sigma,\eta[\ r.\textbf{ref}\ r'\ ]) & \triangleq & \{l\} & \textbf{if } \sigma(r) = \textbf{region}_l \\
\mathcal{A}(\sigma,\eta[\ !r\ ]) & \triangleq & \{l\} & \textbf{if } \sigma(r) = l \triangleright \textbf{ref}\, v \\
\mathcal{A}(\sigma,\eta[\ r := r'\ ]) & \triangleq & \{l\} & \textbf{if } \sigma(r) = l \triangleright \textbf{ref}\, v \\
\mathcal{A}(\sigma,\eta[\ t\ ]) & \triangleq & \{\} & \textbf{if } t \text{ is a different redex form}
\end{array}
$$

Fig. 8. GradCC operational semantics.

GradCC includes one particular type of resource: regions and mutable state, formally representing memory accessible by real Gradient programs. Therefore, if we widen the cv of a program so that it only contains regions, the resulting capture set must give us an upper bound on the regions accessible by the program. We add this property into our correctness theorems as follows.

Theorem 5.4 (Region-aware Progress). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$ such that $\Delta \vdash \mathrm{cv}(t) <: C$ and $\overline{\Delta \vdash r : \mathrm{Reg}\,\hat{}\,\{\textbf{cap}\}}^{\,r \in C}$. Then either there exists r such that $t = r$, or $(\sigma;t) \longrightarrow_e \textbf{\textit{fail}}$, or there exist σ', t' such that $(\sigma;t) \longrightarrow (\sigma';t')$ and $\mathcal{A}(\sigma,t) \subseteq \{l \mid r \in C, \sigma(r) = \textbf{\textit{region}}_l\}$.*

Theorem 5.5 (Region-aware Preservation). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$ such that $\Delta \vdash \mathrm{cv}(t) <: C$ and $\overline{\Delta \vdash r : \mathrm{Reg}\,\hat{}\,\{\textbf{cap}\}}^{\,r \in C}$. Then $(\sigma;t) \longrightarrow (\sigma';t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$ and $\Delta, \Delta' \vdash \mathrm{cv}(t') <: C \cup \{l\}$, where l is the region created during the reduction, if any.*

These theorems are the intended GradCC correctness statements. The technical report proves the following theorems.

Theorem 5.6 (Preservation). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then $(\sigma;t) \longrightarrow (\sigma';t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$.*

Theorem 5.7 (Progress). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then either there exists r such that $t = r$, or there exist σ', t' such that $(\sigma;t) \longrightarrow (\sigma';t')$.*

We only provide an intuitive argument for the soundness of the obscur form **obscur** p **as** x **in** t. We want p to be accessed only in the extent of an **encl** form, because it is improper and may itself access arbitrary capabilities. The cv of an **obscur** form is #. It can only occur in the extent of an **encl** form. Further, x is scoped to this extent (Section 5.1), so accesses to p through x are safe.

6 Evaluation

6.1 Migrating the Scala XML Library

We migrated `scala-xml`, the standard Scala XML library [ScalaXML 2023], to a capture-checked Gradient package. The `scala-xml` library was chosen since most of its code does not need to access any system resources, with the primary exceptions being the XML parser, which may need to resolve DTDs from the filesystem or from the network, and the convenience functions for, e.g., loading and parsing an XML file. Still, capabilities were more common than expected:

- code for rendering an XML object into a `String` manipulated mutable `StringBuffer`-s
- some classes representing XML data had mutable fields, contrary to standard Scala practice
- some functionality was implemented by calling Java code, e.g., parsing XML

The migration also revealed that our formalism should distinguish between records and modules and separate regions from modules. The former allows treating class instances as potentially pure records, as opposed to always-impure (tracked) modules; the latter supports local mutable state.

Despite these difficulties, migrating `scala-xml` to a capture-checked Gradient package required few changes to the codebase. The library has 4200 LoC (excluding comments); adding capture annotations to it required modifying c. 260 LoC and involved no refactoring. Most of the changed lines (c. 200) are similar to the following example: the change involves just a few extra characters.

```
// before the migration
def buildString(sb: StringBuilder): StringBuilder
// after the migration
def buildString(sb: StringBuilder^): StringBuilder^{sb}
```

Another common case was creating a region for local mutable state, which a real implementation would do implicitly. Other notable cases involved objects which store mutable objects in mutable state, which requires making the outer object *region-polymorphic*. A minority of such subtler cases are expected when extending a proof-of-concept design to real-world code.

The `scala-xml` case study shows that migrating a real-world Scala codebase to a non-ocap, capture-checked Gradient package is not a significant amount of effort; such migrations are a valuable intermediate step on the way to migrating a codebase to an ocap module. A full migration may require significantly refactoring the codebase so that it receives all the devices it needs as arguments from its callsites, and will likely require the users of the codebase to adjust their code too. At the same time, ocap modules are more flexible, since they allow their users to *attenuate* the authority of capabilities passed to the module (Section 3.1).

We attach the migrated sources and a detailed report as supplementary material. Both are available in a source code repository. The report lists the steps we took to migrate the library and all the code changes, explains how to understand Scala features like classes and packages in terms of our formalism, and suggests how to verify we migrated the library correctly.

6.2 Implementing Gradual Compartmentalization

We outline the major steps to extending an existing language with gradual compartmentalization.

6.2.1 Add Object Capabilities and Modules as an Extension. We outlined the Gradient support for these features in Section 3. To ensure capability safety, it may be necessary to additionally restrict or *tame* [Miller 2006] existing language features, e.g., an implementation of Gradient would need to tame the Scala standard library (potentially by assigning it appropriate capture signatures) and restrict ocap code from using features such as Java reflection. The necessary work for the Java case was studied by the authors of Joe-E [Mettler et al. 2010] and Wyvern [Melicher 2020]. There are many other examples of ocap extensions for existing languages in the literature, e.g., the Caja extension for Javascript [Miller et al. 2008], the Emily extension for OCaml [Stiegler 2007;

Stiegler and Miller 2006], the CaPerl extension for Perl [Laurie 2007], and the Oz-E extension for Oz [Spiessens and Van Roy 2005].

6.2.2 Track Capabilities in the Type System. Gradient uses Capture Tracking [Boruch-Gruszecki et al. 2023] to track the authority of objects in their types, as we formalised in Section 4 and Section 5. The essential reason for tracking capabilities in types is letting ocap code interact with non-ocap code by using the type system to maintain security guarantees (Section 3).

Non-essential reasons for using Capture Tracking include the borrow safety property (Section 3.2.2), the minimal notational burden of the approach (Section 6.1), and the Capture Tracking extension to Scala 3. Xu and Odersky [2023] describe the algorithmic aspects of the approach.

6.2.3 Support Dynamic Capability Access Restrictions. Gradient's **`enclosed`** blocks rely on runtime support. They can be implemented with, e.g., the LitterBox framework built for Enclosures [Ghosn et al. 2021], which itself relies on hardware assistance. Concretely, an Enclosure is a closure with an access restriction listing packages and system calls accessible by the closure's body. Entering the Enclosure activates the restriction. Nested Enclosures can only increase the restrictions, and they are relaxed by leaving the Enclosure. Enclosures divide the program's memory into disjoint areas (sets of memory pages), each associated with a unique package. Enclosures restrict package access by limiting the accessible memory areas.

A Gradient module instance maps to an Enclosure package (and its memory area). A Gradient device is an object whose methods are language primitives which perform syscalls; it maps to all the syscalls its methods may perform. Thus, an **`enclosed`** block's restriction (which can only mention module instances and devices) maps directly to an Enclosure's restriction and the runtime can enforce it with LitterBox. Hardware assistance allows imposing access restrictions even across FFI calls to binary code which can forge pointers. Furthermore, securing a real-world application (e.g., a web server) via Enclosures has an acceptable performance cost; the slowdown factor can be as small as 1.02 [Ghosn et al. 2021].

6.2.4 Conclusions. Extending an existing language with gradual compartmentalization is an effort of a similar magnitude to implementing a new ocap language. Gradual compartmentalization rests on solid foundations [Boruch-Gruszecki et al. 2023; Ghosn et al. 2021; Melicher 2020; Mettler et al. 2010]; the tasks necessary for implementing it were studied independently for different contexts and each of them is individually well understood.

6.3 Cost Estimation for Dynamic Capability Access Restrictions

In this section, we estimate that implementing the runtime support for **`enclosed`** blocks based on Enclosures would incur an acceptable slowdown. We inspect the performance impact of **`enclosed`** blocks on using `scala-xml` to query an XML dataset. The code "queries" the dataset, always retrieving the same 10 entries to avoid randomness. It parses and converts them to JSON (performing a linear amount of computation and allocation w.r.t. the entry sizes), and writes them to stdout. XML files are parsed under an **`enclosed`** block: `scala-xml` allows using foreign code to parse XML, in which case an **`enclosed`** block allows ensuring that this foreign code does not have excessive privileges (e.g., cannot access the network, to resolve DTDs or otherwise). The JSON library can run without an **`enclosed`** block, since it is implemented in Scala and Capture Tracking can check that the library does not access any sensitive functionality.

Enclosures were implemented using the LitterBox library [Ghosn et al. 2021], itself using Intel processor features (MPK or VT-x)[4]. With LitterBox, changing the current restriction, allocating a memory page associated with a region, and checking if a syscall satisfies the current restriction (if

[4]Intel VT-x, in particular, is a widely available virtualization feature.

any) all take additional time. These costs are hardware-related and do not vary between languages; Ghosn et al. [2021] measured them with a microbenchmark. Enclosures by themselves have an insignificant memory overhead: while they partition the memory into areas, the program needs the same amount of memory. Moreover, areas which are always accessible together are merged, resulting in few (<16) runtime memory areas for typical applications [Ghosn et al. 2021].

We measure the baseline performance by averaging the response time based on 10^6 runs. Separately, we count the events incurring a performance cost, and take the average amount per response. To do so accurately, we run the code on Scala Native in a single thread. In particular, we count the switches between an `enclosed` block and runtime code which needs elevated privileges, as their impact can be significant [Ghosn et al. 2021]; for Scala Native, this only occurs with the GC, which may access arbitrary memory. The benchmark was run on a ThinkPad X1 Carbon 6th gen, with an Intel Core i7-8650U @ 1.90 GHz and 16 GB RAM.

Baseline	MPK raw	MPK slowdown	VT-x raw	VT-x slowdown
1188ms	1195.29ms	1.006x	1207.46ms	1.168x

Each dataset entry is in its own file: the code reads a number of small files in an `enclosed` block, repeatedly paying the syscall cost and performing little computation to offset it. This accounts for nearly (>90%) the entire slowdown. Still, the MPK version of LitterBox would perform near the the baseline, while the VT-x version has a much higher syscall cost. Entering and leaving `enclosed` blocks accounts for the rest of the slowdown. The cost of switching to GC code is insignificant at <15ns per request; moreover, in these measurements the Scala Native runtime enters the GC excessively, as unoptimized code increases the heap size more eagerly and runs the GC a 100x less often. The page allocation cost is particularly minimal at <3ms spread over 10^6 requests on MPK, where it is more costly.

7 Related Work

This section first explores different compartmentalization approaches and highlights their trade-offs, and next discusses the literature on tracking capabilities in types.

7.1 Static Compartmentalization

Object capabilities: There is a long history of research on object capabilities. As early as 1973, Morris described various language features which can support local reasoning about security properties. W7 is an early example of a language with support for capabilities [Rees 1996]. The seminal thesis on the E language [Miller 2006] may have been the first to explicitly recognize and define the object capability approach, as well as provide a detailed description of its benefits. E inspired many other works on restricting existing languages to build a capability-safe subset [Laurie 2007; Mettler et al. 2010; Miller et al. 2008; Spiessens and Van Roy 2005; Stiegler and Miller 2006].

Gradient's approach to modules is very closely inspired by Wyvern [Melicher et al. 2017], which itself is inspired by Newspeak modules [Bracha et al. 2010] and their predecessors, such as MzScheme's Units [Flatt and Felleisen 1998].

Object capabilities together with a module system enable an application to compartmentalize its components and control their access to program and system resources in an intuitive and familiar way. However, they assume that the application's code is uniformly written assuming no ambient authority, which is not true of the vast majority of currently existing code.

Programming Languages: Rust allows circumventing its memory safety guarantees within `unsafe` blocks. The motivation for this feature is that circumventing the guarantees is occasionally necessary for expresiveness and that the blocks themselves can easily be located by tooling. In practice, developers make mistakes: Bae et al. [2021] built a tool for automatically scanning the

Rust ecosystem for vulnerabilities and identified 264 previously unknown memory safety bugs (leading to 76 CVEs). Moreover, combining safe and unsafe languages in a single application can lead to Cross-Language Attacks [Mergendahl et al. 2022], which would have been prevent by the checks of either language alone, static of dynamic. Preventing such vulnerabilities is one reason to only allow executing unsafe code if its behaviour can be dynamically controlled and restricted, as we propose to do in gradual compartmentalization and in Gradient.

PCC & Language Virtual Machines: Proof-carrying code (PCC) [Appel 2001; Necula 1997] is an approach which attaches a formal proof to a software component. The proof is checked at load-time to ensure the component adheres to the desired security policies. Certain compartmentalization solutions, such as domain specific languages (e.g., eBPF [McCanne and Jacobson 1993]), compiler instrumentation (e.g., NaCl [Yee et al. 2009]), or even language virtual machines (e.g., WASM [Haas et al. 2017]) can be see as variations of PCC. While such mechanisms work on the level of bytecode, employing them may still require refactoring code, e.g., eBPF code is required to terminate [McCanne and Jacobson 1993]. They further often target specific ecosystems (e.g., web browsers or kernel module subsystems) and require non-negligible efforts to be adapted to other environments [WASI 2023; WASM-JS 2023; WASM-Web 2023].

7.2 Dynamic Compartmentalization

Processes: Processes are the default mechanism to isolate applications in a time-sharing operating system. They were used to compartmentalize applications such as web browsers [Chromium 2023; Mozilla 2023]. They are a clear boundary around untrusted code that encompasses all the code's resources and has a clear interface to the underlying system to interpose on system calls [Linux 2023]. Further, process-based compartments benefit from supporting arbitrary, pre-compiled binaries.

Most applications, however, assume a shared heap and stack and the ability to directly call their libraries. Compartmentalizing existing applications with processes thus requires heavy refactoring so that untrusted libraries are only directly accessed within a separate process. It incurs non-negligible overheads to turn direct calls into synchronous inter-process communication, requires marshalling arguments between processes, and generally increases resource consumption, either through system metadata or duplication of common code dependencies.

OS abstractions: Several solutions [Bittau et al. 2008; Hsu et al. 2016; Litton et al. 2016] extend operating systems with intra-address-space isolation mechanisms. Light-weight Contexts (lwC) [Litton et al. 2016] let application create intra-process compartments with limited access to the program's resources. Despite being more flexible than processes, such solutions still require modifying applications. As these are generally implemented at the system-level, they do not leverage program-specific semantic knowledge and push the burden of compartmentalization onto the programmer. The lack of a clear migration path to compartmentalized applications may in part explain why none of these solutions made its way into mainstream operating systems.

Hardware Extensions: Application compartmentalization operates at a different spatial and temporal granularity than processes. As a result, hardware security extensions appeared to provide hardware-enforced isolation at either (1) finer-granularity (e.g., Mondrian memory at byte-level [Witchel et al. 2002]), (2) with lower temporal overheads (e.g., Intel Memory Protection [Intel 2020] or VmFunc in Intel VT-x [Uhlig et al. 2005]) to switch between compartments, or (3) both (e.g., CHERI [Woodruff et al. 2014]).

Similarly to OS mechanisms, these solutions require either heavily modifying existing applications, or implementing new software development toolchains [Ghosn et al. 2021; Hedayati et al. 2019; Lind et al. 2017; Vahldiek-Oberwagner et al. 2019] (i.e., compilers, standard libraries, runtime environments). The second approach allows leveraging language or application-specific knowledge

to (partially) automate code compartmentalization, reducing the migration burden. E.g., Enclosures [Ghosn et al. 2021] expose a flexible programming abstraction. They rely on the compiler and the runtime to bridge the gap between programming languages and hardware. The runtime transparently creates compartments and orchestrates transitions. Enforcing isolation via hardware mechanisms allows Enclosures to support heterogeneous environments. Despite their acceptable performance overheads, Enclosures only detect policy violations at run-time. This can lead to costly trial-and-error restriction tuning, which slows down the development process.

7.3 Tracking Capabilities in Types

An effect system can track capability access. For instance, in the region-based memory management system proposed by Tofte and Talpin [1997], an effect system tracks access to regions. Indeed, practically *any* effect system, starting from the seminal work of Lucassen and Gifford [1988], can be used to track capability access. Such systems were also integrated with object capabilities, e.g., Wyvern features an effect system which allows tracking capability access at method granularity.

However, it was observed multiple times (e.g., by Osvald et al. [2016] and Brachthäuser et al. [2022]), that most such systems feature a form of polymorphism which leads to verbose type signatures, which are arguably a key factor impeding their broader adoption [Boruch-Gruszecki et al. 2023]. Capture Tracking instead relies on intuitive capability-based reasoning: capabilities in scope can always be accessed, without needing to state so in types. The types instead track where capabilities are passed, which arguably is also more intuitive. An effect system enforces "asking for permission" to invoke effectful operations, while Capture Tracking types merely clarify which objects may capture capabilities. In practice, Capture Tracking can be retroactively applied to a codebase with a relatively small burden (Section 6).

Systems which track capture of particular tracked objects were proposed: the type-based escape analysis of Hannan [1998] and the Open Closure Types of Scherer and Hoffmann [2013]. Neither system features a lightweight polymorphism mechanism similar to Capture Tracking. Rytz et al. [2012] present a type-and-effect system which assigns simple signatures to higher-order functions. Compared to Capture Tracking, the expressivity of the system is limited: it is impossible to type a function whose result's effect is relative to the function's argument, e.g., function composition.

8 Conclusion

Gradual compartmentalization is a technique for incrementally introducing object capabilities to codebases which provides security benefits even if foreign binary code is present. The approach was applied in Gradient, a proof-of-concept extension to Scala, which has its formal foundations in GradCC. Migrating `scala-xml` showed that Capture Tracking in Gradient allows static authority restriction at the cost of a relatively small degree of effort, providing an intermediate step when introducing object capabilities to a codebase. Thanks to building on proven prior work, implementing Gradient requires acceptable effort, and the performance cost of Enclosures is realistic.

Acknowledgments

This paper was supported by MEYS under the ERC CZ program, grant no. LL2325, and by the SNSF grant no. PCEGP2_186974.

Data-Availability Statement

The migrated `scala-xml` code with the migration report and the benchmark code can be found on GitHub and were published on Zenodo, at abgruszecki/gradient-scala-xml [Boruch-Gruszecki

et al. 2024b] and abgruszecki/gradient-benchmark [Boruch-Gruszecki et al. 2024a] respectively. The accompanying technical report will shortly be released and made available at https://abgruszecki.github.io.

References

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*. Springer, 249–272. ↪ pages 10 and 11

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-Dependent Types. *ACM SIGPLAN Notices* 49, 10 (Oct. 2014), 233–249. https://doi.org/10.1145/2714064.2660216 ↪ page 15

A.W. Appel. 2001. Foundational Proof-Carrying Code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 247–256. https://doi.org/10.1109/LICS.2001.932501 ↪ page 22

Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570 ↪ page 21

Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, Jon Crowcroft and Michael Dahlin (Eds.). USENIX Association, 309–322. http://www.usenix.org/events/nsdi08/tech/full\T1\textbackslash_papers/bittau/bittau.pdf ↪ page 22

Aleksander Boruch-Gruszecki, Adrien Ghosn, Mathias Payer, and Clément Pit-Claudel. 2024a. *Gradient: benchmark code*. https://doi.org/10.5281/zenodo.13385386 ↪ page 24

Aleksander Boruch-Gruszecki, Adrien Ghosn, Mathias Payer, and Clément Pit-Claudel. 2024b. *Gradient: migrated scala-xml*. https://doi.org/10.5281/zenodo.13385375 ↪ page 23

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Transactions on Programming Languages and Systems* (Sept. 2023). https://doi.org/10.1145/3618003 ↪ pages 3, 9, 16, 17, 20, and 23

Aleksander Boruch-Gruszecki, Radosław Waśko, Yichen Xu, and Lionel Parreaux. 2022. A Case for DOT: Theoretical Foundations for Objects with Pattern Matching and GADT-style Reasoning. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 179:1526–179:1555. https://doi.org/10.1145/3563342 ↪ page 15

Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 405–428. https://doi.org/10.1007/978-3-642-14107-2_20 ↪ pages 2 and 21

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-Based Reasoning to Type-Based Reasoning and Back. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 76:1–76:30. https://doi.org/10.1145/3527320 ↪ page 23

Partha Das Chowdhury, Mohammad Tahaei, and Awais Rashid. 2022. Better Call Saltzer & Schroeder: A Retrospective Security Analysis of SolarWinds & Log4j. *CoRR* abs/2211.02341 (2022). https://doi.org/10.48550/arXiv.2211.02341 arXiv:2211.02341 ↪ pages 2 and 5

Chromium. 2023. Chromium sandboxing documentation. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md ↪ page 22

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (1966), 143–155. ↪ page 2

Matthew Flatt and Matthias Felleisen. 1998. Units: Cool Modules for HOT Languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 236–248. https://doi.org/10.1145/277650.277730 ↪ page 21

Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728 ↪ pages 2, 17, 20, 21, 22, and 23

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363 ↪ page 22

John Hannan. 1998. A Type-Based Escape Analysis for Functional Languages. *Journal of Functional Programming* 8, 3 (May 1998), 239–273. https://doi.org/10.1017/S0956796898003025 ↪ page 23

Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review* 22, 4 (Oct. 1988), 36–38. https://doi.org/10.1145/54289.871709 ↪ page 5

Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor ↪ page 22

Raphael Hiesgen, Marcin Nawrocki, Thomas C. Schmidt, and Matthias Wählisch. 2022. The Race to the Vulnerable: Measuring the Log4j Shell Incident. *CoRR* abs/2205.02544 (2022). https://doi.org/10.48550/arXiv.2205.02544 arXiv:2205.02544 ↪ page 5

Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 393–405. https://doi.org/10.1145/2976749.2978327 ↪ page 22

Intel 2020. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel. ↪ page 22

Java. 2021. JEP 411: Deprecate the Security Manager for Removal. https://openjdk.org/jeps/411 ↪ page 3

Ben Laurie. 2007. Safer Scripting Through Precompilation. In *Security Protocols (Lecture Notes in Computer Science)*, Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe (Eds.). Springer, Berlin, Heidelberg, 289–294. https://doi.org/10.1007/978-3-540-77156-2_36 ↪ pages 20 and 21

Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind ↪ page 22

Linux. 2023. Seccomp BPF. https://kernel.org/doc/html/latest/userspace-api/seccomp_filter.html ↪ page 22

James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 49–64. ↪ page 22

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564 ↪ page 23

Guillaume Martres. 2023. Type-Preserving Compilation of Class-Based Languages. (Jan. 2023). https://doi.org/10.5075/epfl-thesis-8218 arXiv:2307.05557 [cs] ↪ page 11

Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. USENIX Association, 259–270. https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet ↪ page 22

Darya Melicher. 2020. *Controlling Module Authority Using Programming Language Design*. Ph. D. Dissertation. Carnegie Mellon University. ↪ pages 2, 4, 19, and 20

Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20 ↪ pages 2 and 21

Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. *Proceedings 2022 Network and Distributed System Security Symposium* (2022). https://doi.org/10.14722/ndss.2022.24078 ↪ page 22

Adrian Mettler, David A. Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*, Vol. 10. 357–374. ↪ pages 19, 20, and 21

Mark Miller. 2006. *Robust Composition: Towards a Unifed Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University. https://jscholarship.library.jhu.edu/handle/1774.2/873 ↪ pages 2, 5, 7, 19, and 21

Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. *Safe Active Content in Sanitized JavaScript*. Google Inc. Technical Report. https://google-code-archive-downloads.storage.googleapis.com/v2/code.google.com/google-caja/caja-spec-2008-06-06.pdf ↪ pages 19 and 21

J. C. Mitchell and R. Harper. 1988. The Essence of ML. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 28–46. https://doi.org/10.1145/73560.73563 ↪ page 10

James H. Morris. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21. https://doi.org/10.1145/361932.361937 ↪ pages 2 and 21

Mozilla. 2023. *Firefox sandboxing documentation*. https://wiki.mozilla.org/Security/Sandbox ↪ page 22

George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. Association for Computing Machinery, New York, NY, USA, 106–119. https://doi.org/10.1145/263699.263712 ↪ page 22

Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You Are What You Include: Large-Scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 736–747. https://doi.org/10.1145/2382196.2382274 ↪ page 1

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification Gone Too Far? Affordable 2Nd-class Values for Fun and (Co-)Effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 234–251. https://doi.org/10.1145/2983990.2984009 ↪ page 23

Vineet Rajani, Deepak Garg, and Tamara Rezk. 2016. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 150–163. https://doi.org/10.1109/CSF.2016.18 ↪ page 5

Marianna Rapoport and Ondřej Lhoták. 2019. A Path To DOT: Formalizing Fully-Path-Dependent Types. *arXiv:1904.07298 [cs]* (April 2019). arXiv:1904.07298 [cs] http://arxiv.org/abs/1904.07298 ↪ pages 14 and 15

Jonathan A. Rees. 1996. *A Security Kernel Based on the Lambda-Calculus*. Technical Report. https://dspace.mit.edu/handle/1721.1/5944 ↪ pages 2 and 21

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP 2012 – Object-Oriented Programming (Lecture Notes in Computer Science)*, James Noble (Ed.). Springer, Berlin, Heidelberg, 258–282. https://doi.org/10.1007/978-3-642-31057-7_13 ↪ page 23

Jerome H. Saltzer. 1974. Protection and the Control of Information Sharing in Multics. *Commun. ACM* 17, 7 (July 1974), 388–402. https://doi.org/10.1145/361011.361067 ↪ page 2

Sandboxdb. 2023. Sandboxdb.org. https://sandboxdb.org ↪ page 3

ScalaXML. 2023. Scala XML: the standard Scala XML library. https://github.com/scala/scala-xml ↪ page 19

Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science)*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 710–726. https://doi.org/10.1007/978-3-642-45221-5_47 ↪ page 23

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. https://www.semanticscholar.org/paper/Gradual-Typing-for-Functional-Languages-Siek/b7ca4b0e6d3119aa341af73964dbe38d341061dd ↪ pages 2 and 15

Fred Spiessens and Peter Van Roy. 2005. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Multiparadigm Programming in Mozart/Oz (Lecture Notes in Computer Science)*, Peter Van Roy (Ed.). Springer, Berlin, Heidelberg, 21–40. https://doi.org/10.1007/978-3-540-31845-3_3 ↪ pages 20 and 21

Marc Stiegler. 2007. Emily: A High Performance Language for Enabling Secure Cooperation. In *Fifth International Conference on Creating, Connecting and Collaborating through Computing (C5 '07)*. 163–169. https://doi.org/10.1109/C5.2007.13 ↪ page 19

Marc Stiegler and Mark Miller. 2006. *How Emily Tamed the Caml*. Hewlett Packard Labs Tech Report. https://www.hpl.hp.com/techreports/2006/HPL-2006-116.pdf ↪ pages 20 and 21

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613 ↪ page 23

R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel Virtualization Technology. *Computer* 38, 5 (May 2005), 48–56. https://doi.org/10.1109/MC.2005.163 ↪ page 22

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner ↪ page 22

Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 309–320. https://doi.org/10.4230/LIPIcs.SNAPL.2015.309 ↪ pages 2 and 15

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.). Springer, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1 ↪ pages 2 and 15

WASI. 2023. Webassembly: WASI. https://github.com/WebAssembly/WASI ↪ page 22

WASM-JS. 2023. Webassembly: JavaScript API. https://webassembly.github.io/spec/js-api/index.html ↪ page 22

WASM-Web. 2023. Webassembly: Web API. https://webassembly.github.io/spec/web-api/index.html ↪ page 22

Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 304–316. https://doi.org/10.1145/605397.605429 ↪ page 22

Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 457–468. https://doi.org/10.1109/ISCA.2014.6853201 ↪ page 22

A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093 ↪ page 14

Yichen Xu and Martin Odersky. 2023. *Formalizing Box Inference for Capture Calculus*. Technical Report arXiv:2306.06496. arXiv. https://doi.org/10.48550/arXiv.2306.06496 arXiv:2306.06496 [cs] ↪ page 20

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted X86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25 ↪ page 22

Received 2024-04-06; accepted 2024-08-18