EPFL

Systems and Formalisms Laboratory

École Polytechnique Fédérale de Lausanne

Implementation of a gradually compartmentalized programming language

by Valentin Aebi

Master Thesis

Prof. Clément Pit-Claudel
École Polytechnique Fédérale de Lausanne
Thesis Advisor

Dr. Aleksander Boruch-Gruszecki
Charles University of Prague
External supervisor

Jun. Prof. Dr. Jonathan Immanuel Brachthäuser
Eberhard Karls University of Tübingen
External Expert

Systems and Formalisms Lab
EPFL IC IINFCOM SYSTEMF
INN 316 (Bâtiment INN)
Station 14
CH-1015 Lausanne

January 2025

Abstract

The object-capability discipline (ocap) is known for enforcing good software engineering practices and helping make software secure and reliable. Ocap code can however not interact seamlessly with non-ocap code without compromising its core properties, which hinders its practical adoption. This thesis builds on previous theoretical work in the field of type systems and proposes a simple programming language, Grattlesnake, that uses capture-checking and runtime checks to provide a gradual approach to ocap. Grattlesnake allows an ocap program to invoke libraries written in its non-ocap sublanguage without compromising the ocap discipline. We show through a case study that our design allows checking interesting properties on non-trivial programs with a very reasonable amount of additional complexity.

Contents

Software artifacts

Compiler, runtime, and agent: `https://github.com/epfl-systemf/grattlesnake-lang`

Case study: `https://github.com/ValentinAebi/sudoku-case-study`

Type-checker for GradCC: `https://github.com/ValentinAebi/Gradient`

This report was built using EPFL Hexhive lab's thesis template, created by Mathias Payer. The source code of the template is available at `https://github.com/HexHive/thesis_template`.

Chapter 1

Introduction

Software projects often consist of a constellation of modules working together. Some of these modules are developed specifically for the project they belong to. But most projects also use third-party libraries [32], which greatly eases software development by saving developers from the need to rewrite the same functionalities again and again. This practice however also comes with its own set of challenges. One of them is to precisely understand the specification of their functionality. Documentation helps, but it is mostly informal, and pretty commonly out-of-date or incorrect [2]. Libraries also often expose vulnerabilities, to the point that the large majority of vulnerabilities in client code are caused by the libraries it depends on [27]. As a result, software components, especially but not limited to third-party ones, may have effects that are hard to reason about and threaten both the safety and the security of software systems. To build trustworthy software, we need ways of controlling such effects.

Dependencies may influence the top-level program in two ways: through the values returned by API methods, and by performing side effects. There exist well-established techniques, like taint-based methods, to control the former [13]. In this thesis, we focus on restricting the latter, namely the side effects of untrusted code, using the object-capability model (ocap). The core idea of ocap is that functions should be able to perform an effect only if they have been given access to a special object, named a *capability*, corresponding to the said effect [11]. For instance, a function that receives the filesystem as an argument is allowed to read and write files, but a function that does not receive the filesystem is not allowed to do so. This differs from the model used by widespread languages like Java that allow any function to read or write files by merely importing the appropriate methods. Similarly, ocap forbids mutable globals like static fields, making it impossible to silently mutate objects that have not been explicitly passed to the function. Ocap thus guarantees that the possible effects of functions are limited by the capabilities that the function is given access to. This makes effects easier to reason about by enabling more fine-grained control on the effects of untrusted functions but also gets rid of documentation problems like a function updating a global without its documentation mentioning it.

Despite its advantages, the ocap discipline has not yet been adopted in practice. One of the reasons is its lack of backward compatibility: allowing an ocap program to invoke non-ocap libraries means giving up the guarantees offered by this discipline, as the said library might just ignore the capability system and perform an effect without asking for permission to do so. A language that statically enforces the ocap model must therefore stay isolated from code written in other languages. This lack of interoperability not only means that the language has to define its own libraries from scratch, but also that migrating a project from another language has to be done at once, which often implies a prohibitive amount of work.

To mitigate this problem and ease the adoption of the ocap model, Boruch-Gruszecki et al. [5] proposed a gradual ocap model that allows statically checked ocap code to invoke non-ocap libraries by relying on runtime checks to make sure that the non-ocap parts of the program do not perform effects that they have not been explicitly allowed to perform. This model is presented as a fictional programming language, Gradient, designed as an ocap extension of Scala. Gradient allows non-ocap code to be invoked in special blocks, named *enclosures*, that enforce dynamic restrictions on the code inside them. Gradient tracks two kinds of resources: devices, which allow accessing system resources like the filesystem or the network, and regions, which are markers for mutable state. It also uses *capture tracking* [6] to account for the possibility of transitively capturing capabilities: an object that captures a capability has to mention this capability in its type, and by doing this it becomes a capability itself.

This thesis is concerned with the implementation of the discipline described for Gradient inside a small programming language. We explore the design and implementation of a simple but expressive language, Grattlesnake, that supports both ocap and non-ocap code and allows a program written according to the ocap discipline to safely invoke non-ocap code. Our research question is the following: does the gradual ocap discipline as described in [5] allow writing interesting programs in a reasonable amount of effort and does it help enforce security and safety properties in these programs? The scope of this project is limited to checking and executing programs entirely written in Grattlesnake (but combining the ocap and non-ocap disciplines), as opposed to supporting compatibility with other languages. Furthermore, checking the programs assumes access to all sources. Grattlesnake is built on top of a small language, Rattlesnake, that I initially designed as a personal project for learning purposes. It is somewhat reminiscent of Kotlin, despite being much simpler and slightly lower-level. Its compiler, written in Scala, targets JVM bytecode.

1.1 Terminology

Grattlesnake is developed in the original Rattlesnake repository on GitHub, where it is designated as Rattlesnake version 0.2-gradient. Artifacts like the compiler and the runtime still use "Rattlesnake" in their name. In this report, we use the expressions "Grattlesnake" and "the original Rattlesnake" to disambiguate between the language developed in this project and the simpler language that it is

based on.

1.2 Contributions

The main contributions of this thesis are as follows:

- Algorithmic formulation of GradCC under the form of a type-checker for a lambda-calculus, along with a pen-and-paper proof that the algorithmic subcapturing rules implemented in the type-checker are equivalent to the declarative formulation by Boruch-Gruszecki et al. [5].
- Design of a language that applies the concepts described in [5], and implementation of this language on top of the Java Virtual Machine. This includes a compiler and a runtime that provides support for the dynamic restriction of device usage and object mutations.
- Evaluation of this design by writing a non-trivial example program in Grattlesnake that uses both ocap and non-ocap code, and proposition of improvements to the design.

Chapter 2

Preliminaries and background

2.1 Methodological note: counting lines of code

The LOC counts mentioned in this report are obtained by counting non-empty lines, where a line is considered empty if it contains only spacing characters. In particular, lines containing a single symbol like a parenthesis or a brace, as well as lines that contain only comments, are part of the count. See the lines counting script for more details: `https://github.com/ValentinAebi/LinesCounter`.

2.2 Capturing types

Capture-checking via capabilities tracked in types is a recently proposed approach to mitigate, among others, the problem of polymorphism in effect systems [6]. It allows marking some objects as capabilities to force the type system to track them. The core principle is that if an object captures a capability, then it has to mention this capability in its capture set. It thus has a non-empty capture set, which makes it a capability itself. For instance (using Scala's syntax):

val f**:** Foo^ = ...
val b**:** Bar^{f} = **new** Bar(f, 42)

The "^" symbol is used to mark `f` as a capability. As `b` captures `f`, its capture set is `{f}`, making `b` a capability as well. Note that the integer value passed to the constructor of `Bar` along with `f` does not need to be mentioned in the capture set as it is not a capability. `Foo^` is a shorthand for `Foo^{cap}`, where `cap` is the root capability, a fictional capability that is used to mark objects as tracked: conceptually, `f` is a capability because it captures `cap`. The formal foundations of capture-checking, namely the $CC_{<:\square}$ calculus, have been studied in [6].

An experimental capture-checking system has been integrated into the Scala programming language and can be enabled by a language import [9]. Foreseen applications of capturing types include checked exceptions (an exception is allowed to be thrown only if the corresponding capability is in scope), but also escape-checking and safer region-based memory allocation [6][9]. In this project, we use them to keep track of system resources (for instance the access to the file system), and mutable state.

2.3 ModCC and GradCC

To model the type system of the Gradient language, Boruch-Gruszecki et al. [5] extend $CC_{<:\Box}$ with graduality constructs that account for the runtime checks. They do so in two steps. The first one is ModCC (figure 2.1), an extension of $CC_{<:\Box}$ supporting records and modules (where a module is a record augmented with an additional field containing a region). The second step is GradCC, which extends ModCC with the actual graduality constructs. In this section, we explain the main concepts of Gradient, ModCC, and GradCC (some of which are already present in $CC_{<:\Box}$).

Subcapturing: The usual subtyping relation works on *shape types*, i.e. types without a capture set. Subcapturing can be thought of as an addition to subtyping that accounts for the capture sets of the types. It is defined as a binary relation $<:$ between capture sets such that $C_1 <: C_2$ if C_2 covers all the elements of C_1, i.e. if a type that captures C_1 is allowed to be treated as a subtype of a type that captures C_2. This is enforced by the CAPT subtyping rule of ModCC (fig. 2.1). In this rule, the first premise is the usual subtyping relation between shapes, the second premise is the subcapturing relation, and the conclusion is capture-aware subtyping. This relation is introduced by $CC_{<:\Box}$, where every $x \in C_1$ of type $T \wedge D$ is covered by C_2 if either $x \in C_2$ or $D <: C_2$. ModCC adds records to the system and modifies the rules accordingly: SC-PATH handles paths rather than variables and SC-MEM says that if one is allowed to capture a record, then one is also allowed to capture any of its fields.

Boxing (also named capture tunneling, introduced by $CC_{<:\Box}$, rules BOX and UNBOX in fig. 2.1): Boxing a term means temporarily hiding the capture set of its type, typically when instantiating a type variable to that type. Before the term can be used, it has to be unboxed, and the unboxing operation requires the variables contained by the capture set to be in the scope where unboxing occurs. For more details about capture tunneling, see section 2.5 in [6]).

Packing (introduced by ModCC, rules PACK, UNPACK, and MODULE in fig. 2.1): When an object captures a local resource, it is usually impossible for the capturing object to outlive the resource. That is, when the resource goes out of scope, the type of the capturing object becomes ill-formed, as the local it depends on is not defined anymore. Packing is a way of circumventing this issue: when the local holding the resource goes out of scope, its occurrence in the type of the capturing object gets replaced by a path from the object to its field (or transitive subfield of one of its fields), using a self-reference, similarly to DOT's recursive types [20].

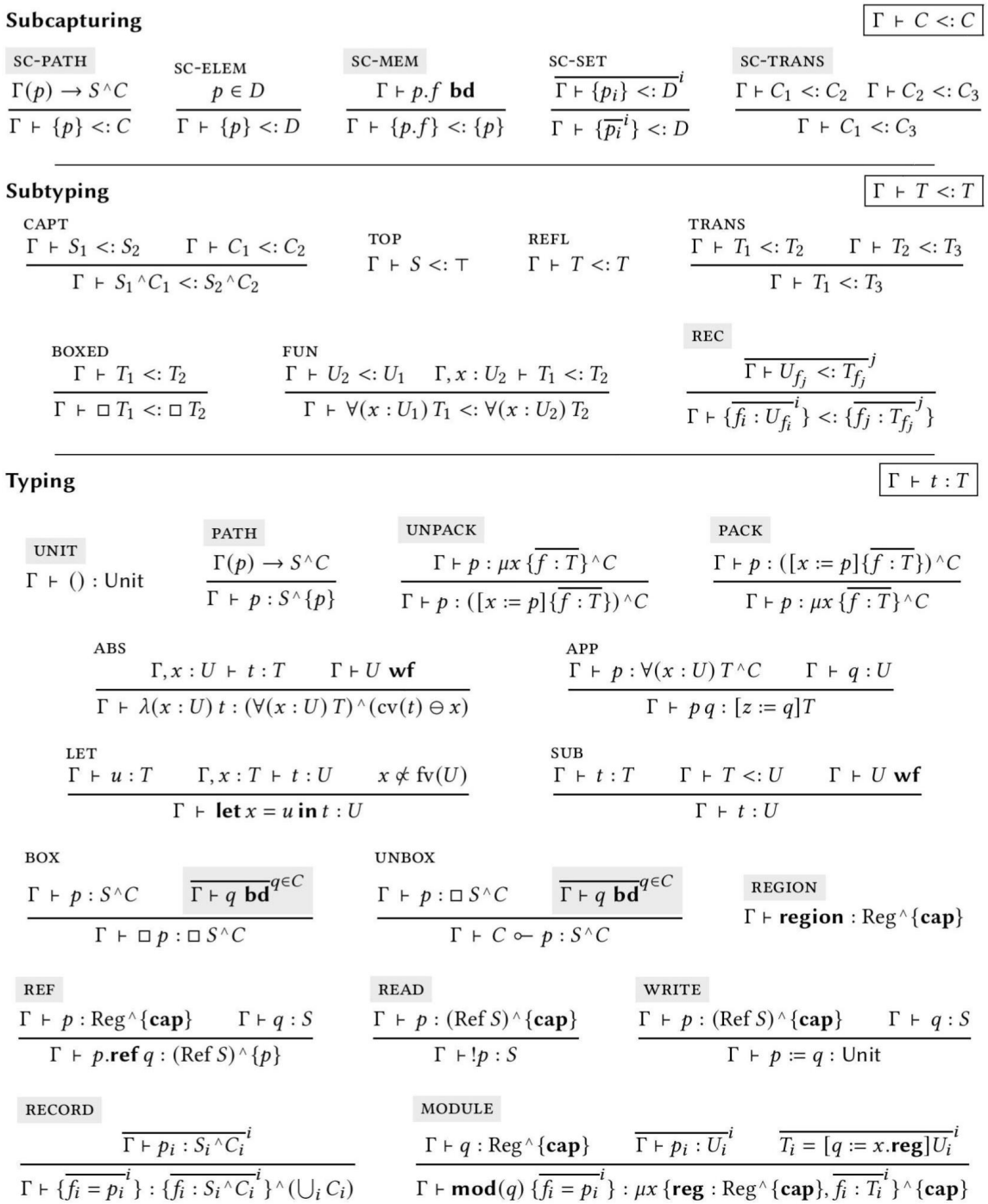

Figure 2.1: ModCC static rules. Highlighted rules and premises are new or changed (resp.) compared to $CC_{<:\Box}$. This figure is taken from [5]. Note the following syntax elements: p stands for path, S and T for type shapes, and C and D for capture sets. $\Gamma(p) \to S\text{^}C_1$ stands for a context lookup (i.e. computing the type of p in the current context, accounting for field selections). $\Gamma \vdash p.f$ **bd** $\iff \exists T.\Gamma(p) \to T$.

The following example shows an invalid term (T is a placeholder for an arbitrary type):

```
let cpb: T^ = ... in
  λu:Unit. cpb
```

This term is invalid because the closure captures cpb and thus cannot escape its scope (unless we upcast it, which is not always desirable). However, the following term, which packs the closure along with cpb in a record, is allowed. Indeed, an application of the PACK rule allows replacing the single element cpb of the capture set of the closure with its path from the recursive qualifier of the record.

```
let cpb: T^ = ... in
let closure = λu:Unit. cpb in
  { cpb = cpb, closure = closure }
```

We omit the types in this example due to their verbosity, but a more complete example of packing, with types (in the notation used by the type checker), can be found in figure 3.2.

At use-site, unpacking simply replaces the recursive qualifier with the unpacked path itself (i.e. if the value of this term is assigned to a variable x, we replace the qualifier with x).

Graduality features (introduced by GradCC, fig. 2.2): Ocap code can invoke non-ocap code only inside enclosures. Values returned from invocations of non-ocap functions have to be marked with a special *capture descriptor*, denoted # and named a *mark*. Such values can only be accessed inside enclosures. The MARK and OBSCUR rules allow communication between ocap and non-ocap code. Obscuring allows a value produced by non-ocap code to be passed to ocap code and can only occur inside an enclosure. The body of enclosures needs to have a pure type, which ensures that no obscured path is leaked from the enclosure.

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

Figure 2.2: GradCC static extensions. Note that R stands for a type shape here. The other syntax elements are as in figure 2.1.

2.4 The Rattlesnake programming language

Note: this section describes the original Rattlesnake language, in the state that served as a starting point to the implementation part of this project.

Rattlesnake is a toy programming language, which I started developing in 2022 as a personal project. It is meant to be simple but relatively expressive and is compiled to Java bytecode by a compiler written in Scala. It supports basic data types like integers, booleans, doubles, chars, and strings, as well as (polymorphic) arrays, C-like nominal structures, and nominal datatypes (i.e. named sets of fields that structures can subtype - they are named interfaces in the original version of the language). Variables can be declared either as `var` (reassignable) or `val` (final). It offers the usual control structures (`if`, `for`- and `while`-loops). Supported expressions include basic unary and binary operators, as well as a length operator and element indexing in arrays and strings. And (`&&`) and or (`||`) operators evaluate the right operand lazily. Returns are performed using a dedicated `return` statement that may be located at any statement position in the function's code.

Rattlesnake also supports casts and type tests with Kotlin-like smart casts. There are no exceptions, but a `panic` statement allows terminating a program with an error message. A function guaranteed to terminate with such a statement may declare `Nothing` as its return type, which is a subtype of all other types. Blocks are statements (i.e. they do not have a result value) and functions that do not return anything declare `Void` as their return type. Constants can be declared at the top level, but they must be strings or primitive types, and their value expression should always be a literal. The language also supports on-demand tail-call elimination: tail calls have to be marked at call site with an exclamation mark between the function name and the arguments list (e.g. `foo!(a, b)`). If a marked call is not in tail position, the compiler rejects the program.

Rattlesnake provides a way of controlling the mutability of objects: for a given struct type or array type `T`, the type system supports two versions of the type: an unmodifiable view (`T`), and a type with mutability permissions (`mut T`). The subtyping relation is such that `mut T <: T`, but not the other way around. The restrictions on modifiability are not transtive: if a variable `s` of (unmodifiable) struct type `S` has a field named `x`, `s.x` may have a mutable type if it is declared as such in the definition of struct `S`. The most useful consequence of this design is the possibility of making unmodifiable array views covariant, which would be unsound with mutable arrays. After this feature proved confusing when interacting with capturing types (especially when an unmodifiable view of a mutable object captures this object's region), it was removed in Grattlesnake. Section 7.4 however proposes a variant of the type system that includes a similar notion of read-only type view.

Regarding the implementation, the compiler uses an LL1 parser without a recovery mechanism, implying that any parsing error immediately terminates the compiler. Errors happening at a later stage are collected and displayed at the end of the phase in which they arose, causing the compiler to terminate. The backend uses the Java ASM bytecode manipulation library [8].

The compiler for the original version of the language, especially its readme that gives more details about its features, can be found at `https://github.com/ValentinAebi/Rattlesnake/tree/rattlesnake-v0.1`.

We think that Rattlesnake is a good platform for our experimental implementation mainly for the following reasons:

- It is simple and its compiler is relatively small (about 6200 LOC, in Scala), which makes it easier to apply profound modifications than in real-world languages.
- Its feature set covers the essentials of real-world languages. Among others: compound types (structures), non-trivial subtyping hierarchies (datatypes), polymorphism (arrays), and mutability both at the level of local variables and in structures.
- It does not have access to system resources (except write access to the console), which allows us to introduce resources in a way that fits the ocap discipline without having to deal with a pre-existing ambient authority.
- Having written every single line of the compiler, I know it well enough to easily apply changes to it and avoid spending too much time understanding bugs, which allows me to focus on the implementation of the gradual ocap model instead of technical issues.

2.5 The Java Virtual Machine

The Java Virtual Machine (JVM) is a runtime system to execute Java bytecode. The bytecode is organized in *class files*, each of which corresponds to a class, interface, or module in the sense of the Java programming language [19]. These classes contain the methods of the program. When a program is executed, the classes are loaded lazily, loading being triggered by the first use of the class. When loading a class, the JVM checks the well-formedness of its bytecode. For example, it uses an abstract interpreter to check that the state of the stack at the time of executing an instruction matches the operand types expected by this instruction [18].

The behavior of a program written in Java bytecode may be modified at load time using bytecode instrumentation. To do so, one must use an agent that intercepts class loads and modifies the bytecode appropriately, resulting in the bytecode that gets executed being different from the one in the class file [1]. A command-line option allows to set up a runtime agent when launching a JVM.

Chapter 3

Exploratory phase: A type-checker for GradCC

Before diving into the implementation of the actual language, the first step of this project consisted in implementing a type-checker for GradCC. The goal was for me to get a better understanding of this formalism, get more familiar with dependent types, and anticipate problems related to the transformation of the rules of ModCC and GradCC into a type-checking algorithm.

3.1 Algorithmic rules for subcapturing

In [5], the subtyping and subcapturing rules of ModCC and GradCC are given in a declarative style. To make them computable, we first need to transform them into an equivalent algorithmic system. We focus on the algorithmic formulation of the subcapturing rules, as the conversion of the subtyping rules into an algorithm is pretty standard and straightforward. Figure 3.1 shows an algorithmic formulation of the subcapturing relation.

3.2 Proof of equivalence to the declarative rules

We show that any judgment that can be proved using one of the systems can also be proved using the other system. We do so by proving theorems 1 and 2.

Theorem 1: ALGO → DECL. *Any judgment provable in the algorithmic system is also provable in the declarative system.*

Proof. Consider a derivation tree of a judgment J in the algorithmic system. We will build a derivation of J in the declarative system as follows. Uses of SC-PATH-ALGO and SC-MEM-ALGO are replaced by

$$\frac{p \in C}{\Gamma \vdash \{p\} <: C}\ \text{SC-ELEM-ALGO}$$

$$\frac{p \notin C \qquad \Gamma(p) \to S\text{^}D \qquad \Gamma \vdash D <: C}{\Gamma \vdash \{p\} <: C}\ \text{SC-PATH-ALGO}$$

$$\frac{p.f \notin C \qquad \Gamma(p.f) \to S\text{^}D \qquad \Gamma \not\vdash D <: C \qquad \Gamma \vdash \{p\} <: C}{\Gamma \vdash \{p.f\} <: C}\ \text{SC-MEM-ALGO}$$

$$\frac{|\{\overline{p_i}^{\,i}\}| \neq 1 \qquad \overline{\Gamma \vdash \{p_i\} <: D}^{\,i}}{\Gamma \vdash \{\overline{p_i}^{\,i}\} <: D}\ \text{SC-SET-ALGO}$$

Figure 3.1: Algorithmic subcapturing rules. Premises in gray account for the order in which the algorithm tries to apply the rules. Premises in blue correspond to the inlining of the transitivity rule (SC-TRANS) of the declarative system.

SC-PATH and SC-MEM (respectively), along with an application of SC-TRANS. SC-ELEM-ALGO is equivalent to SC-ELEM and its uses are left unchanged. Uses of SC-SET-ALGO can also be replaced by SC-SET as both rules have the same conclusion and the single premise of SC-SET is also present in SC-SET-ALGO. The resulting derivation tree is a proof of J in the declarative system. □

To prove the other direction, we need the following lemmas:

Lemma 1: Reflexivity of the subcapturing relation in the algorithmic system. *For all C, $C <: C$.*

Proof. If C is a singleton, then we can use SC-ELEM-ALGO and we are done. Else, SC-SET-ALGO applies, with its premises proven using using SC-ELEM-ALGO. □

Lemma 2. *Let $A = \{\overline{a_i}^{\,i}\}$. Then any derivation tree of $\Gamma \vdash A <: B$ in the algorithmic system includes derivations of $\Gamma \vdash \{a_i\} <: B$ for every i.*

Proof. By case analysis on the last rule used to derive $\Gamma \vdash A <: B$, we see that this is indeed true: by definition for SC-SET-ALGO, and trivially because A is a singleton in the three other cases. □

Theorem 2: DECL → ALGO. *Any judgment provable in the declarative system is also provable in the algorithmic system.*

Proof. Consider a derivation tree of a judgment J in the declarative system. We need to show that there exists a derivation tree for J in the algorithmic system.

The proof goes by structural induction on J. We have the following cases, depending on the last rule used in the derivation of J:

- Case SC-ELEM: replaced by SC-ELEM-ALGO (which is equivalent).

- Case SC-PATH when $p \notin C$: replaced by SC-PATH-ALGO, where the last premise is proved by a derivation of reflexivity, which exists by lemma 1.
- Case SC-PATH when $p \in C$: replaced by SC-ELEM-ALGO.
- Case SC-MEM: The single premise of SC-MEM tells us that there is a type $S \wedge D$ such that $\Gamma(p.f) \to S \wedge D$. Furthermore, $p.f \notin \{p\}$. Thus, either $\Gamma \vdash D <: \{p\}$ in the algorithmic system and we can use SC-PATH-ALGO, or $\Gamma \not\vdash D <: \{p\}$ and we can use SC-MEM-ALGO, where we know by the reflexivity lemma that the last premise can be proved.
- Case SC-SET when $|\{\overline{p_i}^{\,i}\}| = 1$: the premise of the rule is equal to its conclusion, so by IH we are done.
- Case SC-SET when $|\{\overline{p_i}^{\,i}\}| \neq 1$: replaced by SC-SET-ALGO, as we know by IH that the premises can be proved in the algorithmic system.
- Case SC-TRANS: We first show that an application of SC-TRANS can be replaced by applications of the algorithmic rules if the first premise is such that C_1 is a singleton, and then generalize the result. We go by case analysis on the last rule used in the derivation of the first premise of the application of SC-TRANS:
  - Subcase SC-ELEM: We are in the following situation:

$$\dfrac{\dfrac{p \in C_2 \text{ (P1)}}{\Gamma \vdash \{p\} <: C_2}\,\text{SC-ELEM} \qquad \Gamma \vdash C_2 <: C_3 \text{ (P2)}}{\{p\} <: C_3}\,\text{SC-TRANS}$$

    By IH we have that P2 can be derived using the algorithmic rules, implying by lemma 2 that for every $y \in C_2$, we can prove $\Gamma \vdash \{y\} <: C_3$ in the algorithmic system. We know from P1 that $p \in C_2$, allowing us to conclude that $\Gamma \vdash \{p\} <: C_3$ is also provable in the algorithmic system.
  - Subcase SC-PATH when $p \notin C_2$: similar to the case of SC-PATH alone (see above), but with the reflexivity axiom replaced by the second premise of SC-TRANS.
  - Subcase SC-PATH when $p \in C_2$: same as subcase SC-ELEM.
  - Subcase SC-MEM: We are in the following situation:

$$\dfrac{\dfrac{\Gamma(p.f) \to S \wedge D}{\Gamma \vdash \{p.f\} <: \{p\}}\,\text{SC-MEM} \qquad \{p\} <: C_3}{\{p.f\} <: C_3}\,\text{SC-TRANS}$$

    As in the case of SC-MEM alone, the single premise of SC-MEM tells us that $\Gamma(p.f) \to S \wedge D$ for some $S \wedge D$. Furthermore, $p.f \notin \{p\}$. There are two cases. The first one is that $\Gamma \not\vdash D <: \{p\}$, in which case SC-MEM-ALGO applies with the second premise of SC-TRANS as its last premise. W.l.o.g. we can ignore the second case $\Gamma \vdash D <: \{p\}$. Indeed, in this case, the conclusion could equivalently be proved as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma(p.f) \to S\text{^}D}{\Gamma \vdash \{p.f\} <: D}\ \text{SC-PATH} \qquad \Gamma \vdash D <: \{p\}}{\Gamma \vdash \{p.f\} <: \{p\}}\ \text{SC-TRANS} \qquad \{p\} <: C_3}{\{p.f\} <: C_3}\ \text{SC-TRANS}$$

Hence by transforming J we can get rid of this case.

– Subcase SC-SET: w.l.o.g. we can ignore this case. Indeed, it corresponds to the following situation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \{p_i\} <: C_2}\,i}{\Gamma \vdash \{\overline{p_i}^{\,i}\} <: C_2}\ \text{SC-SET} \qquad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash \{\overline{p_i}^{\,i}\} <: C_3}\ \text{SC-TRANS}$$

Note that under our assumption $\{\overline{p_i}^{\,i}\}$ is a singleton, implying that the premise and conclusion of the application of SC-SET are identical. Any proof that makes such use of SC-SET can be simplified into another equivalent proof by removing this step. This of course applies recursively if the premise of SC-SET is itself the conclusion of an application of SC-SET in the derivation tree.

– Subcase SC-TRANS: w.l.o.g. we can ignore this case as well. Indeed, it corresponds to the following situation:

$$\dfrac{\dfrac{\Gamma \vdash C_1 <: C_{12} \qquad \Gamma \vdash C_{12} <: C_2}{\Gamma \vdash C_1 <: C_2}\ \text{SC-TRANS} \qquad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash C_1 <: C_3}\ \text{SC-TRANS}$$

Starting from the same premises, we can derive the same conclusion as follows:

$$\dfrac{\Gamma \vdash C_1 <: C_{12} \qquad \dfrac{\Gamma \vdash C_{12} <: C_2 \qquad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash C_{12} <: C3}\ \text{SC-TRANS}}{\Gamma \vdash C_1 <: C_3}\ \text{SC-TRANS}$$

So by transforming J we can get rid of this situation. Note that this transformation has to happen after the one described in the subcase SC-MEM, which produces left-recursive applications of SC-TRANS as handled by the current subcase.

We now generalize to the case where $C_1 = \{\overline{x_i}^{\,i}\}$ is not a singleton. By IH we know that the derivation of the first premise of the application of SC-TRANS can be rewritten using the algorithmic rules. Thus, lemma 2 tells us that for every i, there is a derivation tree showing $\Gamma \vdash \{x_i\} <: C_2$ in the algorithmic system. Now the singleton case that we just proved applies and allows us to conclude that $\Gamma \vdash \{x_i\} <: C_3$ is provable in the algorithmic system. We can then use SC-SET-ALGO to show that $\Gamma \vdash C_1 <: C_3$.

□

Input term:

```
fn (f: (r: Reg^) → Reg^{r})
   let m =
      let r = region in
      let s = f r in
         mod(r) { me = s }
   in
      m.me
```

Type-checker output:

```
fn (f@0: (r@0 : Reg^{cap}) → Reg^{r@0})
  let m@0 : (self self@0 in { me : Reg^{self@0.reg},
                              reg : Reg^{cap}       })^{cap} =
    let r@1 : Reg^{cap} =
      region
    in
      let s@0 : Reg^{r@1} =
        (f@0 r@1) : Reg^{r@1}
      in
        mod(r@1) { me : Reg^{self@0.reg} = s@0 }
  in
    m@0.me
 : ??

———————————————————— 1 error(s) ————————————————————————————
[examples/ex7.gradcc:3:5] forbidden capture: let body has type
Reg^{m@0.reg}, which depends on let—bound variable m@0
    let m =
    ^
```

Figure 3.2: GradCC term and corresponding type-checker output (formatted to fit this figure). `fn` stands for λ and defines an abstraction. `self` defines a recursive qualifier (like μ in the formalism). The module packs its `me` field with the region captured by it. Unpacking occurs in the body of the outermost let, which triggers an error as it causes the type of the body of the let to depend on variable `m` defined by this very let.

3.3 Implementing the type-checker

The GradCC type-checker is a standalone Scala program that takes as input a file containing a GradCC term and outputs a typed representation of this term (i.e. a formatted version of the term along with type annotations for some of the subterms), as well as the list of type errors that have been found during type-checking, if any. It consists of a parser, followed by a renaming phase (that assigns disambiguation indices to the variables), the type-checking phase itself, and a pretty-printer that displays the typed version of the term. The whole software is ~2200 LOC, with ~800 LOC for the type-checking phase itself (including its auxiliaries). The core of the type-checker is a pattern matching on all possible terms, that computes their type according to the typing rules of GradCC. In some cases, it may assign a type to a term even though one of its subterms is not typeable, if the type of the term does not depend on the subterm whose type computation failed (e.g. an assignment to a reference always has type `Unit`). As in the formalism, the type-checker expects terms in A-normal form. Figure 3.2 shows an (invalid) GradCC term along with the output of the type-checker when given this term.

3.4 Experimenting with the type-checker: observations

This type-checker is minimal in the sense that it can only type-check the language defined by a strict translation of the GradCC rules as described in [5], implying that it does not even support integers or booleans. This implementation did not reveal major issues with the practical applicability of the type system. It however revealed that the PACKING rule can lead to some ambiguities when the path that needs to be replaced by packing is assigned to more than one record field. Figure 3.3 shows an example of such a situation.

```
let rc =
    let r = region in
    let f = fn (u: Unit) r in
        { a = r, b = r, f = f }
in
    { x = rc.a, f = rc.f }
```

Figure 3.3: Example term showing that packing is not completely deterministic. In the first record, the type-checker has two possible ways of packing the field `f`: its captured variable `r` can be replaced by either of fields `a` and `b`. If one chooses `a`, then also the second record can be packed successfully, and the whole term type-checks. However, if one chooses `b`, then the packing of the second record fails, and the overall term is rejected.

Chapter 4

Design of the Grattlesnake gradual ocap language

The core part of this project is a compiler and a JVM-based runtime system for Grattlesnake, a gradually compartmentalized version of Rattlesnake based on Gradient and GradCC as described in [5]. The design goals are as follows:

- Grattlesnake should support both ocap and non-ocap code, and no piece of code should be allowed to exceed the capabilities granted to it, no matter whether non-ocap code is involved or not.
- The language should be simple to make experimentation easier, but it should include enough features to be representative of real-world languages.
- It should be expressive enough so that one can write meaningful non-trivial programs with a reasonable amount of effort.

4.1 Language modes

Grattlesnake can be thought of as the union of two languages: one that follows the ocap discipline and uses capture-checking, and another that has no support for capturing types and where access to devices is not restricted. The former is the default one. To enable the latter, a source code file must start with the `#nocap;` directive. All definitions in a file share the same language mode. In this report, we will refer to these language modes as "ocap mode" (default mode, files without a `#nocap;` directive) and "non-ocap mode" (files starting with a `#nocap;` directive).

4.2 Resources: devices and regions

The core mechanism for Grattlesnake to control effects is resource tracking, the principle being that any object that has captured a resource has to mention it in its type, either directly or transitively. Just like in Gradient, we track devices and regions.

Two devices are offered: the filesystem and the console. The filesystem, referred to using the `fs` keyword, enables to read or write from/to a file and to create or delete files and directories. The runtime maintains a collection of open files that are identified by their filehandle (an integer). Grattlesnake code must use this filehandle, returned by the function that opens files, to refer to the associated file when communicating with the filesystem. Figure 4.1 shows an example. The console, represented by the `console` keyword, enables reading and writing from/to the command line console. It is worth noting that for simplicity, other devices like the network (which is mentioned on several occasions in Gradient example programs in [5]), or the serial port, are not supported by Grattlesnake.

```
val fh = fs.openW("path/to/file.txt");
fs.write(fh, "Hello world");
fs.close(fh);
```

Figure 4.1: Example of code that involves the filesystem. Note the use of `openW` which opens the file in write mode, as opposed to `openA` that opens a file in append mode and `openR` that opens a file in read mode.

Regions are markers for mutable state. Every mutable object declared in ocap code has to be associated with a region so that the runtime can enforce dynamic mutability restrictions when the object is passed to non-ocap code. No region can be created by non-ocap code. These regions differ from the ones used in region-based memory allocation in that the language has no support for region deallocation and still needs garbage collection for the whole heap. A region is nothing else than a permission that may be passed to an enclosure to allow the code executed in it to modify the objects mapped to that region. Some implementations may still combine both concepts by allocating objects in a memory region that corresponds to their region in the sense of GradCC. Grattlesnake however does not do that and its type-system has not been designed to ensure a safe deallocation of regions.

Ocap code tracks regions and devices using capture-tracking. Non-ocap code does not use capture sets (which, from the point of view of the type system, are replaced with a special capture descriptor, `#`, the mark of GradCC), but may be subject to the dynamic restrictions set up by the ocap code that invoked it.

4.3 Packages and modules

Along the lines of Gradient, Grattlesnake's top-level definitions include modules and packages. Module declarations have to mention a (possibly empty) list of imports, that is very similar to primary constructor declarations in Kotlin or Scala. Imports can be of three sorts:

- Parametric imports: much like classes in OO languages like Kotlin or Scala, modules can store values of any inhabited type that are passed to their constructor.
- Package imports: a module can import a package to enable its functions to use that package. Package imports are only supported by ocap modules. They are introduced by the `package` keyword, preceded by # if the imported package is a non-ocap one.
- Device imports: just like packages, devices are imported by the constructor of the module, using the `device` keyword. Only ocap modules can have such imports.

For instance, a module `Logger` that is parametrized with the path to the log file and requires access to a non-ocap package named `UnsafeLogger` and direct access to the filesystem could be declared as follows:

```
module Logger(logFileName: String, #package UnsafeLogger, device fs) {
    ...
}
```

The compiler synthesizes a constructor that takes as explicit arguments only the parametric imports. The imported packages and devices are imported implicitly, meaning that the compiler will report an error at instantiation site if the module imports a package or device that is not allowed in the environment in which the instantiation occurs. The capture set of the instantiated module mentions the imported packages and devices, but also any explicit argument that happens to be a resource itself. If the argument is not a stable path (i.e. an expression that will provably always return the same value if evaluated several times), its contribution to the capture set of the module is its own capture set. An instantiation of the `Logger` defined above could look like this:

```
val logger: Logger^{UnsafeLogger,fs} = new Logger("log.txt")
```

Packages are simpler in that they do not have an explicit import list. They thus cannot be parametrized, but ocap packages still import packages and devices implicitly by merely referring to them in the code of their functions. For example, the following package imports the console and thus mentions it in its capture set:

```
package ConsolePrinter {
   fn printInt(i: Int){ console.print(i as String) }
}
```

Ocap packages must also follow the ocap discipline regarding their imports: the compiler allows a package A to import another package B only if B does not import A (either directly or transitively). Packages thus do not allow cyclic imports and conceptually follow the discipline described in Gradient: a package is a singleton module whose instantiation occurs before the execution of the entry point of the program and after the instantiation of its dependencies.

Modules and packages are referenced by the keyword `me` in their body (which is an equivalent of the `this` pointer of other languages like Java). Invocations of functions in the same module or package can omit the `me` keyword, but accesses to fields cannot. Modules and packages provide encapsulation: module fields (corresponding to parametric imports) are internal to the module instance, and both packages and modules can declare private functions, which can be accessed by any instance of the same module or package, but not more.

4.4 Structs and datatypes

Grattlesnake supports structs, which are unencapsulated groupings of named fields. A struct is declared using the `struct` keyword followed by a type name and the list of struct fields. The compiler synthesizes a constructor that takes an argument for every field, meaning that structs must be completely initialized at creation. Structs may or may not be mutable (depending on whether their declaration starts with the `mut` keyword), and in mutable structs, only the fields whose declaration is prefixed with `var` are reassignable. Structs may subtype datatypes, which specify a set of fields that all implementing structs must own. Datatypes may also subtype other datatypes, as long as this creates no cycle in the subtyping relation. In all cases, the fields of the subtype must be a superset of the fields of the supertype, and the types of the fields are covariant if the field is immutable and invariant if the field can be reassigned. Subtyping between a struct and a datatype, or between two datatypes, may occur only if both are defined in the same source file. The capture set of a field may mention the fields preceding it in the list of fields (except for reassignable fields), but not the fields that come after it. This is similar to the way Scala treats capturing types in class fields, and can be seen as a restricted version of the PACKING rule of ModCC. Figure 4.2 shows an example of a simple datatype with structs subtyping it.

```
datatype IntList
struct Nil : IntList
struct IntCons : IntList { head: Int, tail: IntList }
```

Figure 4.2: Definition of a datatype representing lists of integers

Mutable structs must be associated with a region at instantiation (if the instantiation happens in ocap code). To do so, one has to use `new@r` instead of `new` as the instantiation operator, where `r` can

```
datatype MutList

mut struct MutCons : MutList {
    var head: Int,
    tail: MutList^{reg}
}

struct Nil : MutList
```

Figure 4.3: Example hierarchy involving a mutable struct

be replaced by any expression of shape `Region`. The region associated with a mutable struct can be referred to as a special field of the struct, named `reg` (and similar to the `reg` field of modules in ModCC). This special field may only be used in capture sets (i.e. it is not permitted to read the `reg` field of an object to e.g. instantiate a new object in that region). This restriction is motivated by the fact that not all mutable objects are associated with a region (as they may have been instantiated by non-ocap code). The intended usage of the `reg` field is to allow a regular field to capture the region associated with the struct without the need for an explicit region field that exists at runtime. See figure 4.3 for an example.

As this example suggests, a mutable struct may subtype a datatype not marked as mutable, but not the other way around (partially because a datatype that is marked as mutable must ensure that all its subtypes have a `reg` field).

The computation of the capture set of structs is similar to the computation of the capture set of modules in that the resource values assigned to non-reassignable fields by the constructor appear in the capture set. Regarding the reassignable fields, their contribution to the capture set consists of the captures declared by the field type itself, minus the ones that refer to the preceding fields of the struct, as those have already been accounted for by the contributions of the previous fields to the capture set.

4.5 Arrays

Arrays are mutable, polymorphic in their element type, and invariant. They are very similar to the references of GradCC, which can be seen as arrays of length 1. They can be created in two ways: either by providing only the length and letting the runtime initialize them to the default value (0 or null), or by explicitly giving all elements. Note that as null is not supported in Grattlesnake, trying to load from an array at an index whose value has not been initialized results in a runtime error. Arrays need to be associated with a region (we assume the existence of a region `r` in the example below):

```
val xs = arr@r Int[10];  // array of 10 cells, all initialized to 0
val ys = [1, 2, 3]@r;    // array of 3 cells
```

```
datatype Result
struct Success : Result { value: Int }
struct Failure : Result { errorMsg: String }
```

```
fn printResult(res: Result) {
    val s: String;
    if res is Success {
        s = "succeeded with a result value of " + res.value as String;
    } else if res is Failure {
        s = "failed: " + res.errorMsg;
    };
    console.print(s + "\n");
}
```

Figure 4.4: Pattern matching example, on the `Result` datatype. When analyzing `printResult`, the compiler detects that all possible `Results` are covered by the two branches of the if and that no else branch is necessary. It deduces that `s` on the last line has been initialized in all cases when the control flow reaches that point, and accepts this code.

The element type is boxed, implying that the only value captured by an array (from the point of view of the type system) is its region. Following the typing rules of GradCC (and even $CC_{<:\square}$) that prevent a root-capturing type from being unboxed, the compiler reports an error if the element type of an array captures the root capability.

4.6 Casts, smart-casts, and pattern matching

Grattlesnake supports downcasts, which are performed using the `as` keyword. Downcasts only apply to type shapes, it preserves the capture descriptor of the cast value. The `as` keyword can also be used for type conversions, for instance, to convert between primitive types, or to convert a primitive type to its string representation.

Like in Kotlin, Grattlesnake's smart casts allow branches of an if-statement or the ternary operator whose condition is a type test on a local value to temporarily treat the local as a value of the type it has been tested against. For instance, given the `IntLists` defined in figure 4.2, the following expression type-checks:

```
val headOrDefault = when ls is IntCons then ls.head else 0
```

To enable a kind of pattern matching, the compiler is also able to detect when the branches of an if-statement have covered all possible structs for a given datatype (example in figure 4.4).

4.7 Restricted blocks

Restricted blocks allow statically ensuring that some code block does not access more resources than what the programmer intended. Restricted statements take a capture set and the code block in which we want capability access to be restricted. The free variables of the code block must be covered by the capture set to be accepted. This is primarily useful when using (ocap) packages, as their capture set is defined implicitly. In Grattlesnake, restricted blocks are statements (as opposed to expressions), meaning that they do not have a result value. If a value needs to be extracted from such a block, it has to be assigned to a variable (or late-initialized value) that is defined in a scope external to the restricted block. Figure 4.5 shows an example of a restricted block that limits the capabilities of a package and returns a value to the outer scope.

```
val command: String;
restricted {console} {
    command = ConsoleReader.readLine();
}
... // do something with the command
```

Figure 4.5: Example of a restricted block. Assuming that `ConsoleReader` is an ocap package, this ensures that this package (and thus the `readLine` method invoked on it) does not have access to a device other than the console.

4.8 Enclosures, graduality, and runtime checks

Enclosures are syntactically very similar to restricted blocks: they also take a capture set and a block of executable code. Semantically, they differ in that the checks for capabilities are dynamic instead of static. When the execution flow enters an enclosure, the runtime sets up an environment that forbids the use of capabilities other than the ones mentioned in the capture set. Capture sets that are part of enclosures differ from other capture sets in that the values mentioned in them can only be devices or regions (since it must be possible to check their usage at runtime), and are evaluated during program execution (as opposed to being present only in types). If the enclosed code exceeds the capabilities granted to it, e.g. by modifying an object whose region is not mentioned in the capture set, then the runtime triggers the termination of the program, preventing the illegal operation from taking place.

It is also worth noting that while the static discipline works by controlling *access* to capabilities, the dynamic one controls the *use* of capabilities. This means for instance that code inside an enclosure whose capture set does not mention the filesystem is allowed to create an instance of a struct that stores the filesystem in one of its fields. What it is not allowed to do is to use the filesystem, e.g. by writing to a file. This motivates the notion of marking: as we have no control over what a

value outputted by non-ocap code may have captured, we mark its type by letting it capture the mark # instead of a usual capture set. Values of a marked type are allowed to be the receiver of a function call only if this call happens inside an enclosure. The purpose of this restriction is that no matter what capabilities the marked value has captured, their usage will still be subject to dynamic restrictions enforced by the enclosure. For instance, in the following snippet, any attempt from `UnsafeLogger` to access a device other than the filesystem will result in an error that terminates the program:

```
enclosed {fs} {
    UnsafeLogger.log("Hello");
}
```

Like restricted blocks, enclosures are statements, and the standard way of escaping a value from an enclosure is to assign that value to a late-initialized value defined in the outer scope. A marked value is allowed to live outside of an enclosure, as long as no function call is made on it.

GradCC provides two ways to convert between marked types and regular capturing types: marking (MARK rule) and obscuring (OBSCUR rule). Marking in Grattlesnake is very similar to GradCC: a marking operator # allows one to mark a value before passing it to a non-ocap function. Obscuring is implemented as an extension of the subcapturing relation to allow `T^# <: T^` only inside enclosures. The following restrictions guarantee that the obscured value of type `T^` cannot escape the enclosure (and thus that no code that is not subject to the dynamic restrictions enforced by the enclosure can access the obscured value):

- Variables and late-initialized values are not allowed to capture the root capability, hence an obscured path cannot be assigned to them.
- Array elements and reassignable struct fields are not allowed to capture the root capability, hence the obscured value cannot be stored in them.
- Returns are forbidden inside an enclosure, hence the obscured value cannot be returned. This restriction also guarantees that the execution flow reaches the end of the enclosure, where the compiler inserts instructions that reset the dynamic restrictions, before moving on to code outside of the enclosure.

Chapter 5

Implementation of a compiler and runtime for gradual ocap programs

Compiling and executing Grattlesnake programs requires two components: a compiler and a runtime. The starting point for the implementation of the compiler was of course the compiler of the original Rattlesnake language. Regarding the runtime, we use the Java Virtual Machine (JVM). This choice has the advantage of providing a garbage collector and being a relatively simple compilation target thanks to the high-level nature of Java bytecode. Furthermore, it is the compilation target of the original Rattlesnake language, meaning that we can reuse most of the backend, as well as the primary compilation target of Scala, which Gradient is an extension of. On top of the JVM, a runtime implemented as part of this project offers a dynamic-checks-aware implementation of the two devices supported by Grattlesnake (the filesystem and the console), and primitives for region management.

5.1 Runtime and agent

The runtime component is implemented in Java. It assumes single-threaded execution, which is always the case as Grattlesnake has no concurrency primitive. It consists of a main runtime class that handles regions and stores the permissions, two classes for the two devices, and an exception class meant to be thrown when code run inside an enclosure exceeds its permissions.

5.1.1 Regions

Regions are merely identifiers, implemented as integers. Creating a new region means incrementing a global counter to get a new identifier. The mapping from mutable objects to regions is imple-

mented as a `WeakHashMap` to not prevent the objects from being garbage collected. An alternative solution would have been to add a region field to the bytecode representation of mutable structures, but this would not generalize to arrays. Despite the inefficiency of querying a hash map every time an environment needs to be checked, this solution has been chosen for this experimental implementation because of its simplicity and because it fits well into the model of associating mutable objects to regions only if the object is instantiated in ocap code.

5.1.2 Environments

Environments are implemented as a stack, in which every frame represents an environment set up by an enclosure. The stack is empty at startup, and every execution of an enclosure pushes a new environment when entering the enclosure and pops it when exiting the enclosure. This implementation allows to account for nested enclosures. While it probably makes little sense to nest two enclosures in the same function, nested calls interleaving ocap and non-ocap functions may cause several environments to be on the stack at the same time. Of course, the current environment is defined by the top frame, which is always a subset of the deeper frames (otherwise the evaluation of the capture set of the nested enclosure would have crashed the program). Environments store the allowed regions in a set and contain a flag for every device, telling whether it is allowed to be used or not.

5.1.3 Devices

The console device provides two methods: `print`, which takes a string as an argument and prints it to the console, and `readLine`, which reads a line from stdin. Before executing, these methods query the environment, if any, to make sure that they are allowed to execute.

The implementation of the filesystem device is slightly more complex as the device class is stateful. It maps file handles to `FileWriters` for files open in write or append mode, and `FileReaders` for files open in read mode. The `close` method removes the reader or writer corresponding to the provided filehandle from the map containing it and closes it.

5.1.4 fastutil

The various collections used in the runtime classes (with the notable exception of the `WeakHashMap` mentioned above) are instances of collections of the *fastutil* framework [14], an API of type-specific collections that do not box their elements of primitive types as usual collections do. This is meant to reduce the memory footprint of the runtime and possibly improve its performance in code that makes intensive use of dynamically-checked environments or devices. No measure has however been made to confirm this gain.

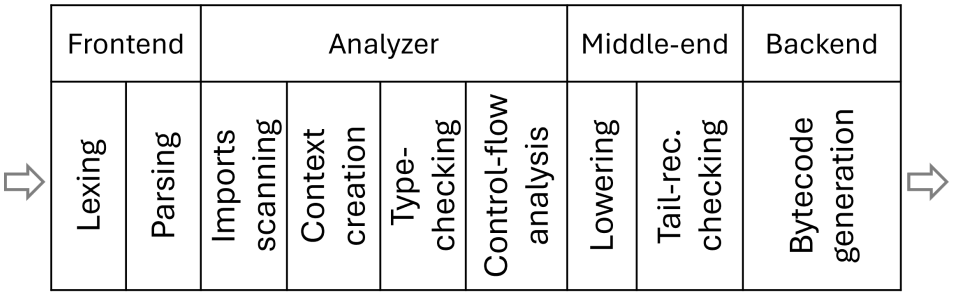

Figure 5.1: Compiler phases

5.1.5 Agent

The runtime must check that the current environment allows objects to be modified before performing a modification on them. While the current design, which does not support modular compilation, would enable the compiler to directly insert these checks during compilation, I chose to still use an agent to instrument the code and add these checks in the bytecode only when loading it from the class files. This choice has two main motivations. First, interoperability with arbitrary class files is a pretty obvious step on the road to making this design usable in practice (I considered supporting interoperability as part of this project, but lacked the time to implement it), hence it makes sense to already show that an agent can perform this instrumentation without access to the source code. Second, this design is closer to the one described for Gradient in [5], where these checks are performed by the runtime and not encoded in the program by the compiler.

5.2 Compiler

This section is a description of the successive compilation phases that the compiler uses to transform Grattlesnake source code into executable Java bytecode. Figure 5.1 is a diagrammatic representation of these phases.

5.2.1 Frontend: scanner and parser

The frontend is made of a scanner (named lexer in the code) followed by an LL1 parser. Few changes have been made to these compared to the original compiler, except for the addition of the new syntax forms and a few bug fixes. The scanner and parser are somewhat inspired by Silex and Scallion [12], two libraries for lexing and parsing developed by EPFL LARA lab. However, unlike Scallion, the parser of Grattlesnake detects LL1 conflicts only dynamically, e.g. when the next token matches both branches of a disjunction. The parser has no recovery mechanism: any parsing error terminates the whole compiler. While this is pretty drastic and would probably be unacceptable for

a real-world implementation, it is worth noting that it caused no significant issue in this project, both when writing test programs and when developing the case study. Therefore, I did not put effort into adding a recovery mechanism.

5.2.2 Analyzer: imports scanner, context creator, type-checker, and control-flow analyzer

One of the core parts of this project was to add capture-checking capabilities to the type-checker of Grattlesnake. However, computing captures (or even types) requires knowledge of the types declared in the program. Gathering this knowledge is the role of the context creator, a phase that traverses the definitions of modules and packages and stores the signatures in a context object (named `AnalysisContext`) that will be passed to the later phases of the compiler. This context creation phase also traverses the code of packages to find occurrences of devices and other packages and add them to the signature of the package. A challenge here is to also detect the implicit uses of devices and packages by the instantiation of modules that import the said devices and packages. To avoid fixpoint computations, the easiest solution I found was to add another phase, the imports scanner, to the pipeline, before the context creator. This phase merely creates a map from every module to the devices and packages that it imports and passes this map to the context creator. The context creator also uses a cycles finder to detect cyclic subtyping between datatypes and dependency cycles between packages that are supposed to be ocap.

Regarding the type-checker itself, it computes the type of every expression, including its capture set, and writes it into the AST node corresponding to the expression, so that the later phases have access to the type of expressions. It also resolves function calls, detects smart casts, and writes a map from local values to their smart-casted type into the AST so that the backend knows when to apply these casts (as smart casts have to be explicit in the bytecode). The type-checker works on the original variable identifiers, it is not preceded by a renaming phase. To make this safe, scoping rules prevent shadowing. These rules are also enforced by the type-checker. This explains why modules must use the `me` keyword to refer to their fields, as fields are allowed to have the same name as locals.

The control flow analyzer runs after the type-checker. It has two main roles:

- Ensuring that all variables and late-initialized values are always initialized before their first use and that values are never assigned more than once.
- Ensuring that non-`Void` functions never reach the end of their body without encountering a return or panic statement. This includes figuring out if a sequence of if statements whose conditions involve type tests and that have no terminating else cover all possible subtypes of the type of a local value. In the positive case, the analysis can ignore the missing else branch, which enables a form of pattern matching, as described in section 4.4. This information is also written into the AST to inform the lowering phase that it must generate a fake else branch

(since the bytecode verifier run by the JVM is unable to detect that no else branch is needed).

It performs these checks by traversing the AST and simulating the execution of program instructions along the edges of the (conceptual) control-flow graph. Explicitly constructing the control-flow graph is not required as Grattlesnake supports no complex control-flow instruction whose target needs to be resolved like gotos or breaks.

5.2.3 Middle-end: lowerer and tail-recursions checker

The lowering phase is the only one that rewrites the AST. Its role is to transform some complex constructs like for loops into simpler ones like while loops, to reduce the number of constructs to be handled by the backend. Some of these transformations are conditioned to flags set in AST nodes by previous phases. The lowering phase of the Grattlesnake compiler is rather aggressive and not followed by an optimizer, which sometimes results in convoluted bytecode. This is inherited from the original compiler and, while for the scope of this project we do not care much about the implications on the efficiency of the generated bytecode, a rather annoying consequence of this is that decompiling the generated class files to Java code sometimes produces overcomplicated code (this happens e.g. when opening a class file in the Intellij IDE).

The tail-recursions checker merely checks that all function calls annotated as tail calls are indeed the very last instruction of their function, so that they can be eliminated by the backend (the fact that these calls must refer to the enclosing function is already checked by the type-checker). This check happens after the lowering phase because lowering clarifies the execution order of instructions, particularly in cases like `a && b` that gets lowered to `when a then b else false`, allowing `b` to be a tail-recursive call if the `&&` operation is itself in tail position.

5.2.4 Backend: JVM bytecode generator

The last phase of the compiler transforms the desugared AST into bytecode. Modules are compiled to regular classes. Packages are compiled to singleton classes, with a single static field containing the singleton instance. Instructions are inserted into instantiations of mutable structs and arrays to invoke functions in the runtime that save the mapping from the struct or array to its region. References to packages are compiled to instructions that load the singleton field of the class representing the package. References to devices, which are also implemented as singleton classes, work similarly.

Among others, the backend also eliminates tail calls marked for elimination, adds non-nullity checks on array loads to keep the language free of null pointers, and compiles string conversions from numeric values (like `0 as String`) to invocations of the corresponding method of the Java standard library (`Integer.toString(int)` in this example).

Chapter 6

Evaluation

6.1 Functional correctness

While the relatively small size of the Grattlesnake language makes its compiler significantly simpler than the ones of real-world languages, it is still complex software (~7300 LOC, in Scala, involving non-trivial algorithms). Due to its very experimental nature, it would certainly be presumptuous to affirm that it is stable and reliable. However, a test suite of approximately 90 tests check most aspects of its functionality and prevent regressions. Among these tests, ~40 are faulty programs with formalized comments indicating where the compiler is supposed to report errors. The testing system parses these comments and compares their location and error messages to the ones of the issues reported by the compiler. ~45 tests involve well-formed programs that the compiler must compile and execute. The test runner then compares the program output (namely data written to the console and/or the stack trace in case of a program that is expected to crash) to the expected output. Overall, the test suite has a line coverage of ~84% on the whole compiler and ~88% on the type-checker alone.

The implementation of the case study has also been a test for the reliability of the system. I found a few bugs during this implementation, which led to the addition of reproducers to the test suite. Some of these bugs were enough to make the compiler unsound from the point of view of the type system, but none of them required deep changes to the compiler or runtime.

6.2 Case study: a sudoku solver in Grattlesnake

I implemented a simple but complete Grattlesnake program as a case study, to evaluate the language and its type system. The case study is a sudoku solver, which reads a grid from a file, solves it

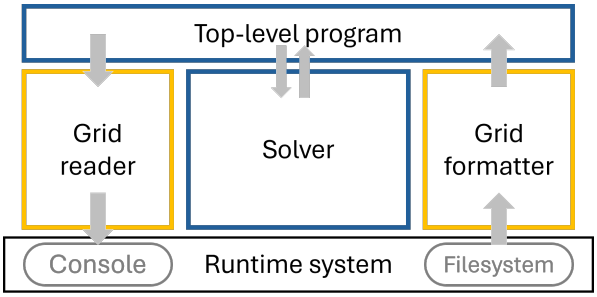

Figure 6.1: Architecture of the solver. Program parts in orange are implemented without following the ocap discipline, whereas the top-level program and the solver package (in blue) are made of statically checked ocap code. Arrows indicate information flows.

according to the rules of the sudoku game, and displays the result on the console. The code for the case study can be found at `https://github.com/ValentinAebi/sudoku-case-study` (the readme contains a link to a Wikipedia page with the rules of the sudoku game).

6.2.1 Architecture of the solver

Note: the explanations in this section will probably make a lot more sense if the reader looks at the code at the same time (`src` directory in the repo of the case study, in particular the files `Main.rsn` that contains the main function and `Grid.rsn` that contains the `Grid` module).

The sudoku solver is written in ocap mode but uses two non-ocap "libraries", represented by source files marked with the `#nocap` directive (`TableReader.rsn` and `GridFormatter.rsn`). The solver itself can be viewed as an ocap library, invoked from the main code as well. Figure 6.1 shows a diagrammatic representation of the architecture. The resolution algorithm is based on candidate elimination for empty cells, with guessing and backtracking when the elimination process gets stuck.

One of these libraries is a grid reader: it reads a file and expects a grid in this file in the form of a grid of chars. To make the setup more realistic, the library is designed in a "grid-agnostic" way. That is, it simply outputs a two-dimensional array of characters without any knowledge of its interpretation as a sudoku. Only then the top-level program transforms it into a numeric sudoku grid.

This numeric grid is then passed to the solver, implemented as a package, whose invocation happens in a restricted block that statically ensures that the solver does not access devices. This can be useful, for instance, to make sure than no debug print has been forgotten in the code, as this would cause the solver package to capture the console and thus exceed the capability granted to it by the restricted block.

The grid itself is implemented as a module that wraps the two-dimensional array of `Cells` representing the grid with its two regions: one of them is a mutability permission on the "frame" of the grid, that is its array structure. Code that has this permission can modify the grid structure, like replacing a row of cells or swapping two cell objects. The other region gives access to the values. A piece of code that has access to the values region is allowed to update the values of the cells (that is, the cell's value in the sense of the game, as well as the array representing the remaining candidates for empty cells). In the case where one would want to pass this cell to a non-ocap solver (which does not happen here), one could run the said solver in an enclosure whose capture set would only contain the region that allows to modify the values, so that the solver can update the values but not tamper with the structure of the grid.

We also want to display the grid on the console before and after solving it. To do so, we use a second non-ocap "library", which, given a grid as a two-dimensional array of `Chars`, displays it with a separation between the sectors of the requested dimensions (3x3 in the case of our sudoku grid). This library is run in an enclosure that gives it access only to the console.

6.2.2 Observations

This example program showcases the key elements of the gradual ocap discipline: static restrictions using restricted blocks, dynamic restrictions using enclosures, and data transfers between ocap and non-ocap parts of the program, in both directions. It also uses both regions and devices. One weakness might be that it uses enclosures only to allow access to devices, and restricted blocks only to grant access to mutable state. Checking a larger number of combinations would however probably require either a larger example program, or a less natural design.

The gradual ocap discipline did not feel like a major productivity killer when implementing this program. I probably spent more time fixing the solver algorithm than refactoring my code to make it compliant with the type system, and this despite using a fairly simple algorithm that I had already implemented in other occasions.

However, a pain point was the conversion of two-dimensional arrays from non-ocap to ocap and back, which required defensive copies. Regarding grid reading, I had to copy the grid, of type `arr^# arr^# Char`, into a new grid of type `arr^{r} arr^{r} Char`, expected by the `mkGrid` function that creates the grid module. Also note the need for `mkGrid` to be region-polymorphic in the region `r`, as taking a value of type `arr^ arr^ Char` as its grid argument would mean capturing the root capability in an array element type (capturing the root only on the outer array type would be fine, though, although less precise). Similarly, when converting the grid back to display it using non-ocap code (again as a two-dimensional array of `Chars`), I had to make a shallow copy of the array. The reason why the copy has to be deep in the former case and may be only shallow in the latter is due to the capture set of the element type of the desired array type: the root capability in the former case, which is not allowed to appear in the capture set of an array element type as opposed to the mark in

the latter case, which is allowed to appear in the capture set of an array element type thanks to the UNBOX-MARK rule of GradCC (which allows the unboxing of such an element type).

Chapter 7

Future work

7.1 Missing features: OOP, FP, and cross-language interoperability

7.1.1 Closures and generics

Grattlesnake is a minimal language, and the addition of some features could both enhance its expressiveness and make it closer to real-world languages. One of these features is closures, which not only allow to factorize and reuse more code but also have interesting interactions with capture-checking because of the free variables that they capture. Unfortunately, time was lacking to implement closures as part of this project, although I expect that this would not be very hard to do. Another desirable feature would be genericity in contexts other than arrays, which, together with closures, is one of the building blocks of APIs that use capture-checking in Scala.

Regarding closures, an important point would be their interaction with the local (mutable) variables that they capture. Like in Scala, captured variables would need to be written on the heap rather than on the stack, by storing them in a cell object. This would make these variables heap-allocated mutable memory and require them to be associated with a region. A possible syntax for closures and region-associated variables is sketched in figure 7.1.

```
val r: Region^ = newreg;
var @r capturedVar: Int = ...;     // associate capturedVar with region r
val closure: (Int) ->{capturedVar} Int = fn (x: Int) -> {
    capturedVar += x;
    return capturedVar;  // exit the closure only
}
```

Figure 7.1: A possible syntax for closures and heap-allocated `vars`.

7.1.2 Object-oriented programming

Grattlesnake provides two essential features of object-oriented languages: encapsulation, with packages and modules, and subtyping between user-defined types, with structs and datatypes. It does however not provide a concept that ties them together like classes do in object-oriented languages. It could thus be interesting to add interfaces to the language, which modules and packages could implement. This has been in the to-do list of this project for some time, but could not be implemented because of lack of time. Another lacking feature that could probably be implemented reasonably easily is (shallow) mutable modules (by the means of reassignable fields inside them). This would tie together encapsulation and mutability, a combination that does not currently exist on its own right in Grattlesnake, even though a module whose parameters include arrays or mutable structs is effectively a mutable module.

7.1.3 Code packaging and interoperability with other JVM-based languages

A more difficult but very important improvement direction would be to support interoperability with other (non-ocap) JVM-based languages. A real-world implementation of the gradual ocap model will very probably never become widely adopted if it does not provide backward compatibility with other languages, as the very point of introducing graduality in the ocap discipline is to enable a gradual migration of existing codebases.

To fully support interoperability with other JVM-based languages and make it possible to export compiled and possibly even obfuscated capture-checked libraries, it would be a great advantage to be able to check that code that claims to be ocap indeed is without the need to access its sources. A solution to do so could be to propagate capture sets throughout compilation until the backend (which the compiler already does) and encode them in the bytecode in the form of persisted annotations.

7.2 Runtime system and performance

During this project, I focused on correctness rather than performance. The only optimizations used in this project are early exit from runtime methods when no environment is set and the use of *fastutil* to limit the number of boxing operations in the runtime (see paragraph 5.1.4). In particular, the heavy transformations performed by the lowering phase to limit the number of constructs to be handled in the backend lead to convoluted and probably inefficient bytecode. Hence, adding an optimization phase to the compiler would make a lot of sense. This seems like a necessary condition before one can profile or benchmark the code to estimate the performance drop due to runtime tests since doing so with the current implementation is at risk of giving results that are biased by the inefficiency of the rest of the bytecode. It is also worth thinking about using a modified JVM

instead of an agent, which might improve performance, but also make it easier to provide security guarantees.

7.3 Automatic unmarking

Some types, particularly unencapsulated ones, are guaranteed to never capture any region or device, be it directly or transitively. This is for instance the case of primitive types like integers or strings. The compiler is aware of this and automatically unmarks such types when they are returned by non-ocap code.

It would however be possible to unmark more types than what the compiler currently does. For example, an instance of a structure whose fields are all integers never needs a mark, as no instance of this structure will ever give access to a region or device. The compiler does not perform unmarking in such cases, but it would probably be reasonably easy to modify it so that it does. It is worth noting that a solution to this problem in the general case requires to take into account possible cycles (e.g. a struct `A` containing a field that is a struct `B`, which itself has a field of type `A`), and possible subtypes if unmarking is to also apply to datatypes.

7.4 Permissions granularity and read-only types

GradCC is designed in such a way that the permissions granted to statically-checked ocap code can be more fine-grained than the ones granted to non-ocap code by an enclosure. Indeed, static checking allows e.g. to grant access to a particular instance of a logger module, whereas an enclosure can not be more precise than allowing access to the filesystem. Similarly, in a statically-checked setting, one may allow a module to read from the console but not output data to it. One would do so by passing to it a module that wraps the filesystem and only exposes functions for reading, along the lines of the membrane pattern described in [23].

There is however a case where dynamic checks can be more precise than static ones: preventing the mutation of regions. Indeed, while dynamic checks detect mutations, static checking cannot do better than controlling what code has access to mutable objects. Even if the mutable object is wrapped in a membrane that does not allow to mutate it, the capture set of the membrane still has to account for the transitive capture of the region.

To solve this problem and allow statically-checked read-only accesses to mutable objects, one can consider the following design. Instead of directly capturing the root capability, regions would capture a special capability, which we will call `mut`, that represents the "region root" and itself captures the root capability `cap`. The language could then offer a type wrapper that allows exposing an arbitrary mutable object as a read-only object, e.g. `readonly S` for `S` a mutable struct type. Then,

the following subtyping rule would allow to "drop" regions from the capture set, e.g. when passing the object to a function that does not need to modify it:

$$\frac{\Gamma \vdash C' <: \{\texttt{mut}\}}{\Gamma \vdash \texttt{T\^{}}\big(C \cup C'\big) <: \text{readonly } \texttt{T\^{}}C}$$

It would however be necessary to formally analyze the consequences of this modification of the type system to make sure that it does not make it unsound. Of course, the read-only permission would have to be transitive: one should not be able to read a mutable reference to an object out of a read-only reference. One expected issue with this design is bad interactions with regions-based memory management if capture-checking is used to guarantee the safety of regions (as described as an extension of capture-calculus in [4]): if regions are not only permissions to mutate objects but also real memory areas that need to be deallocated, then being able to drop regions allows the read-only view on the object to escape the scope of the object's region and may lead to a dangling pointer.

7.5 A gradual ocap real-world language?

In this project, we took a "bottom-up" approach. Starting from a toy language with a very reduced feature set, we designed a more complex language whose entire design was done with ocap graduality in mind. We think that this approach was well-suited to what we wanted to do: figure out if it is possible to write meaningful programs in a gradual ocap language without too many issues. In theory at least, continuing in this direction *could* eventually lead to a language with a feature set that would be rich enough to be adoptable in practice.

Another approach would be to start from a full-blown language and add support for the gradual ocap discipline to it. That is the approach chosen for the Gradient language described in [5], which extends Scala. This probably necessitates more work before the language can be used in demonstrative toy scenarios but may be a better approach in the long run if the goal is a practically usable language (even though getting rid of the ambient authority granted by the language to every module is likely pretty challenging). Considering this, a natural direction for further work could be to adapt the type system of Gradient and Grattlesnake to a real-world language. The obvious candidate is Scala since it already has a capture-checker and Gradient's design is based on Scala, but other, possibly simpler, languages might be worth considering.

Chapter 8

Related Work

Note: this chapter is partially inspired from the Related work *section of [5].*

8.1 Ocap languages

Several languages have been proposed that enforce the ocap discipline. One of them is Wyvern [25]. Similar to Gradient and Grattlesnake, Wyvern features a system of composable modules, where a module needs to be instantiated to an object with explicitly passed capabilities before one can use it [22]. This slightly differs from the design of Grattlesnake, where devices and packages may be passed implicitly instead of appearing in the list of arguments passed to the constructor of the module. Wyvern has also been used for a security-oriented case study aiming among others to assess the practicality of ocap [15].

The E language [23] uses capabilities to enforce the principle of least authority. E allows interfacing with Java using a special capability that functions need to have access to before they can invoke Java methods. Some classes from the Java standard API have also been audited to determine their safety level and the safest ones require a less powerful capability to be invoked.

Pony [10], an actors-based programming language, uses unforgeable tokens to represent access to system resources. Pony's safety may however not hold when invoking C code from a Pony program (which is allowed by the foreign function interface). To mitigate that risk, the Pony compiler accepts arguments that specify which packages are allowed to use the foreign function interface and runtime checks prevent other packages than the specified ones from using it.

Newspeak [7] also uses object capabilities. In particular, it develops the principle of treating modules as objects. It goes further than the current version of Grattlesnake in this direction, e.g. by supporting modules that implement interfaces.

The Monte [24] Python-like programming language is built around the principle of least authority and uses object capabilities to enforce it.

8.2 Dynamic compartmentalization

Various techniques have also been proposed for dynamic program compartmentalization. One of them is Enclosures [16], a new language construct that makes it possible for programmers to precisely decide what capabilities they grant to libraries invoked by their code. The authors claim that enclosures could be added to most programming languages, and describe implementations for both Go and Python where they enforce the restrictions using the LITTERBOX framework, that relies on hardware mechanisms. The `enclosed` block introduced by Gradient and implemented in Grattlesnake is an example of an implementation of such an enclosure, its main additional feature being the interaction with capture tracking.

Another interesting recent work is SECOMP [28], an extension of the CompCert verified C compiler with support for isolated compartments. SECOMP intends to mitigate the vulnerabilities due to undefined behavior in C libraries, but currently limits the data transfer between compartments to scalar data.

The CHERI [30] hardware compartmentalization system enables an incremental adoption of object capabilities and compartmentalization in C, to the price of relying entirely on dynamic checks.

8.3 Type-based capability tracking

Being able to track capabilities of various sorts is the motivation of the capture tracking project in Scala [6]. Scala however provides capabilities tracking as a tool for API designers to guarantee various properties, which differs from the approach taken in this project, as what we want to guarantee is language properties rather than API properties. This is however not the first work done on type systems that control captures. For instance, the previously proposed Open closure types [26] are closure types that specify which information their instances capture from their defining environment.

A type system aiming to allow finer-grained capability control than the usual ocap model has recently been proposed in [31]. Its purpose is to allow interfaces to mark some of their operations with states such as unavailable or optional, to make it possible to restrict the capabilities of an object before passing it to a function. This system aims to ultimately support interaction with third-party code.

8.4 Graduality in type systems and programming languages

Graduality is a recurring topic in programming language design. One of the contexts to which graduality can apply is gradual types, which allow migrating a codebase to a safer static discipline in a gradual manner thanks to their compatibility with dynamically typed code. TypeScript [3], a gradually typed extension of JavaScript, is an example of a widely used gradual language. It is however worth noting that TypeScript does not perform automatic runtime checks, meaning that the responsibility of dynamically checking types remains with the programmer. Other projects aiming to retrofit a gradual type system into a dynamically typed language include Reticulated Python [29] and Typed Racket [17], which uses runtime checks at the border between typed and untyped code.

Graduality and language modes can also apply to topics other than types and can be part of a language since its inception, especially when the static discipline enforced by the language is restrictive. For instance, Rust's [21] unsafe sublanguage allows specific parts of a program to not obey borrow-checking, which increases the expressiveness of the language to the cost of having to trust programmers for using the unsafe mode responsibly.

Chapter 9

Conclusion

In this project, we implemented a programming language with gradual support for the ocap model, using the ModCC type-system described in [5]. We show how the type-system can interact with the runtime to provide the desired properties. We also give an example of a non-trivial program written in this language in which only a minor additional effort is necessary to leverage the type system and enforce interesting properties about the capabilities granted to invoked libraries. This experiment suggests that gradual compartmentalization via object capabilities does not have a prohibitive complexity and has the potential to become widely used in future mainstream languages, or future versions of current ones.

Grattlesnake is a rather small language, which barely covers more than the very essential concepts of ocap, graduality, and general-purpose programming. Adding more features to it would make it easier to build more complex examples and observe how the gradual ocap discipline interacts with more realistic programs. We hope that the observations made during this project will help guide future work about ocap language design and that Grattlesnake will further help in designing experiments about this and related topics, thanks to its simplicity that makes it fairly easy to modify.

Bibliography

[1] https://docs.oracle.com/javase/10/docs/api/java/lang/instrument/package-summary.html. Accessed: 2025-01-03.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. "Software Documentation Issues Unveiled". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[3] Gavin Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9.

[4] Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. *Tracking Captured Variables in Types*. 2021. arXiv: 2105.11896 [cs.PL]. URL: https://arxiv.org/abs/2105.11896.

[5] Aleksander Boruch-Gruszecki, Adrien Ghosn, Mathias Payer, and Clément Pit-Claudel. "Gradient: Gradual Compartmentalization via Object Capabilities Tracked in Types". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689751. URL: https://doi.org/10.1145/3689751.

[6] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. "Capturing Types". In: *ACM Trans. Program. Lang. Syst.* 45.4 (Nov. 2023). ISSN: 0164-0925. DOI: 10.1145/3618003. URL: https://doi.org/10.1145/3618003.

[7] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. "Modules as Objects in Newspeak". In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Ed. by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 405–428. DOI: 10.1007/978-3-642-14107-2_20. URL: https://doi.org/10.1007/978-3-642-14107-2%5C_20.

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. "ASM: A code manipulation tool to implement adaptable systems". In: (Jan. 2002).

[9] *Capture Checking*. https://docs.scala-lang.org/scala3/reference/experimental/cc.html. Accessed: 2024-12-31.

[10] Sylvan Clebsch. "'Pony': co-designing a type system and a runtime". PhD thesis. Imperial College London, 2017.

[11] Dominique Devriese, Lars Birkedal, and Frank Piessens. "Reasoning about Object Capabilities with Logical Relations and Effect Parametricity". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016, pp. 147–162. DOI: 10.1109/EuroSP.2016.22.

[12] Romain Edelmann, Jad Hamza, and Viktor Kunčak. "Zippy LL(1) parsing with derivatives". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 1036–1051. ISBN: 9781450376136. DOI: 10.1145/3385412.3385992. URL: https://doi.org/10.1145/3385412.3385992.

[13] Pär Emanuelsson and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools". In: *Electronic Notes in Theoretical Computer Science* 217 (2008). Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008), pp. 5–21. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.06.039. URL: https://www.sciencedirect.com/science/article/pii/S1571066108003824.

[14] *fastutil: Fast & compact type-specific collections for Java™*. https://fastutil.di.unimi.it/.

[15] Jennifer A. Fish, Darya Melicher, and Jonathan Aldrich. "A case study in language-based security: building an I/O library for Wyvern". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 34–47. ISBN: 9781450381789. DOI: 10.1145/3426428.3426913. URL: https://doi.org/10.1145/3426428.3426913.

[16] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. "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. Virtual, USA: Association for Computing Machinery, 2021, pp. 255–267. ISBN: 9781450383172. DOI: 10.1145/3445814.3446728. URL: https://doi.org/10.1145/3445814.3446728.

[17] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. "A Transient Semantics for Typed Racket". In: *Art Sci. Eng. Program.* 6 (2021), p. 9. URL: https://api.semanticscholar.org/CorpusID:237511479.

[18] Xavier Leroy. "Java Bytecode Verification: Algorithms and Formalizations". In: *J. Autom. Reason.* 30.3–4 (Aug. 2003), pp. 235–269. ISSN: 0168-7433. DOI: 10.1023/A:1025055424017. URL: https://doi.org/10.1023/A:1025055424017.

[19] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java® Virtual Machine Specification - Java SE 23 Edition*. https://docs.oracle.com/javase/specs/jvms/se23/html/index.html. Accessed: 2024-12-31.

[20] Sam Lindley, Conor McBride, Phil Trinder, Don Sannella, Nada Amin, Karl Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. "The Essence of Dependent Object Types". In: *Lecture Notes in Computer Science* 9600 (Mar. 2016). DOI: 10.1007/978-3-319-30936-1_14.

[21] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: https://doi.org/10.1145/2692956.2663188.

[22] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. "A Capability-Based Module System for Authority Control". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Ed. by Peter Müller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 20:1–20:27. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.20. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2017.20.

[23] Mark Samuel Miller. "Robust composition: towards a unified approach to access control and concurrency control". AAI3245526. PhD thesis. USA, 2006.

[24] *Monte*. https://www.monte-language.org/. Accessed: 2025-01-22.

[25] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. "Wyvern: a simple, typed, and pure object-oriented language". In: *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and InHerItance*. MASPEGHI '13. Montpellier, France: Association for Computing Machinery, 2013, pp. 9–16. ISBN: 9781450320467. DOI: 10.1145/2489828.2489830. URL: https://doi.org/10.1145/2489828.2489830.

[26] Gabriel Scherer and Jan Hoffmann. "Tracking Data-Flow with Open Closure Types". In: *Lecture Notes in Computer Science*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Stellenbosch, South Africa: Springer Verlag, Dec. 2013, pp. 710–726. URL: https://inria.hal.science/hal-00911656.

[27] Yijun Shen, Xiang Gao, Hailong Sun, and Yu Guo. "Understanding vulnerabilities in software supply chains". In: *Empirical Software Engineering* 30.1 (Nov. 2024), p. 20. ISSN: 1573-7616. DOI: 10.1007/s10664-024-10581-2. URL: https://doi.org/10.1007/s10664-024-10581-2.

[28] Jérémy Thibault, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Catalin Hritcu, and Andrew Tolmach. *SECOMP: Formally Secure Compilation of Compartmentalized C Programs*. 2025. arXiv: 2401.16277 [cs.PL]. URL: https://arxiv.org/abs/2401.16277.

[29] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. "Design and evaluation of gradual typing for python". In: *SIGPLAN Not.* 50.2 (Oct. 2014), pp. 45–56. ISSN: 0362-1340. DOI: 10.1145/2775052.2661101. URL: https://doi.org/10.1145/2775052.2661101.

[30] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 20–37. DOI: 10.1109/SP.2015.9.

[31] Roland Wismüller, Damian Ludwig, and Felix Breitweiser. "Extending the Object-Capability Model with Fine-Grained Type-Based Capabilities." In: *The Journal of Object Technology* 23 (Jan. 2024), 1:1. DOI: 10.5381/jot.2024.23.1.a1.

[32] Asimina Zaimi, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos. "An Empirical Study on the Reuse of Third-Party Libraries in Open-Source Software Development". In: *Proceedings of the 7th Balkan Conference on Informatics Conference*. BCI '15. Craiova, Romania: Association for Computing Machinery, 2015. ISBN: 9781450333351. DOI: 10.1145/2801081.2801087. URL: https://doi.org/10.1145/2801081.2801087.