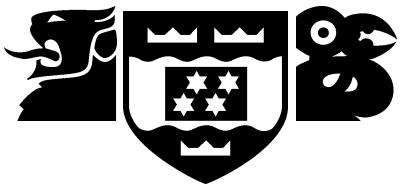
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Capability-Flavoured Effects

Aaron Craig

Supervisor: Alex Potanin, Lindsay Groves, Jonathan Aldrich (CMU)

Submitted in partial fulfilment of the requirements for Bachelor of Science with Honours in Computer Science.

Abstract

Many modern applications require developers to build safe systems out of potentially unsafe components, but existing languages are insufficient in the techniques they provide for identifying untrustworthy or unsafe code. This report explores how capability-safety enables a low-overhead effect-system by introducing a capability calculus `CC`. The way in which it can help developers make more informed decisions about whether to trust code is illustrated with several examples.

Acknowledgments

I would like to give thanks to the following people.

- Alex Potanin, Jonathan Aldrich, and Lindsay Groves — for being wonderful supervisors and giving their endless wisdom and support.
- Darya Kurilova and Julian Mackay — for feedback and suggestions on the formalisms.
- Ewan Moshi — for his friendship and support over the years.
- My family — especially my parents, Mary and John, and my sisters, Amber and Rachel.

Contents

Chapter 1

Introduction

Good software is distinguished from bad software by qualities such as security, maintainability, and performance. We are interested in how the design of a programming language and its type system can make it easier to write secure software.

There are different situations where we may not trust code. One example is in a development environment adhering to ideas of *code ownership*, wherein developers may exercise responsibility for specific components of the system [2]. When a developer writes code to interact with another component, they can make false assumptions about how it should be used. This can break correctness or leave components in a malconfigured state, putting the whole system at risk. Another setting involves applications which allow third-party plugins, some of which could be malicious. Such an example is a web mash-up, which brings together several disparate web applications into one unified service. In both cases, despite the presence of untrustworthy components, we want the system to function securely.

It is difficult to determine if a piece of code should be trusted, but a range of approaches can be taken about the issue. One is to *sandbox* untrusted code inside a virtual environment. If anything goes wrong, damage is theoretically limited to the virtual environment, but in practice there are many vulnerabilities [6, 14, 28, 24]. Verification gives a comprehensive analysis of the behaviour of code, but the techniques are heavyweight and developers must have a deep understanding of how they work in order to use them [10]. Furthermore, verification requires a complete specification of the system, which may be undefined or evolving throughout the development process. Lightweight analyses, such as type systems, are easy for the developer to use, but existing languages are insufficient in the tools they provide for detecting and isolating untrustworthy components [4, 27]. A qualitative approach might be taken, where software is developed according to best-practice guidelines. One such guideline is the *principle of least authority*: that software components should only have access to the information and resources necessary for their purpose [22]. For example, a logger module, which only needs to append to a file, should not be given arbitrary read-write access. Another is *privilege separation*, where the division of a program into components is informed by what resources are needed and how they are to be propagated [23]. This report focuses on the class of lightweight analyses, and in particular how type systems can be used to reject unsafe programs and put developers in a more informed position to make qualitative assessments.

One approach to privilege separation is the capability model. A *capability* is an unforgeable token granting its bearer permission to perform some operation [7]. For example, a system resource like a file or socket can only be used through a capability granting operations on it. Capabilities also encapsulate the source of *effects*, which describe intensional details about the way in which a program executes [17]. For example, a logger might `append` to a `File`, and so executing its code would incur a `File.append` effect. In the capability model,

this requires the logger to possess a capability granting it the ability to append to that particular file.

Although the idea of a capability is an old one, there has been recent interest in its application to programming language design. Miller has identified the ways in which capabilities should proliferate to encourage *robust composition* — a set of ideas summarised by "only connectivity begets connectivity" [16]. As a result, a program's components are explicitly parameterised by the capabilities they may use; the only effects a component can incur are those for which it has been given a capability. Building on these ideas, Maffeis et. al. formalised the notion of a *capability-safe* language and showed a subset of Caja (a JavaScript implementation) is capability-safe [15]. Another capability-safe language, and one we use in this report, is Wyvern [18].

Effect systems were introduced by Lucassen and Gifford to determine what code can be safely parallelised [13]. They have also been applied to problems such as determining which functions might be invoked in a program [26] or determining which regions in memory may be accessed or updated [25]. Knowing what effects a piece of code might incur allows a developer to determine if code is trustworthy before executing it. This can be qualitatively assessed by comparing the static approximation of its effects to its expected least authority — a "logger" implementation which could write to a `Socket` is not to be trusted!

Despite these benefits, effect systems have seen little use in mainstream programming languages. Rytz et. al. believe the verbosity of their annotations is the main reason [21]. Successive works have focussed on reducing the developer overhead through techniques such as inference. Inference enables developers to rapidly prototype without annotations, incrementally adding them as their safety needed, but the benefit of capabilities for this has received less attention: because capabilities encapsulate the source of effects, and because capability-safety impose constraints on how they can propagate through the system, the effects of a piece of code can be safely approximated by inspecting what capabilities are passed into it. This is the key contribution of this report: capability-safety facilitates a lightweight, incremental effect discipline.

We begin by discussing preliminary concepts involving the formal definition of programming languages, effect systems, and Miller's capability model. Chapter 3 introduces the Operation Calculus `OC`, a typed lambda calculus with a simple notion of capabilities and their operations in which all code is effect-annotated. Relaxing this requirement, we then introduce the Capability Calculus `CC`, which permits the nesting of unannotated code inside annotated code in a controlled, capability-safe manner. A safe inference about the unannotated code can be made by inspecting the capabilities passed into it from its annotated surroundings. In chapter 4 we show how `CC` can model practical examples. We finish with a summary and a literature comparison.

Chapter 2

Background

In this chapter we cover the necessary concepts and existing work informing this report. First we detail how a programming language and its type system are formally defined, and how to prove the type system is correct. For this purpose, we present a toy language called Expression-Based Language (`EBL`). We then summarise a variant of the simply-typed lambda calculus $\lambda^{\rightarrow}$. $\lambda^{\rightarrow}$ is the basis for many programming languages, including the capability calculus `CC`. `CC` is also a capability-based language with an effect system. To understand what this means we cover some existing work on effect systems and discuss Miller's capability model.

2.1 Formally Defining a Programming Language

A programming language can be defined by giving three sets of rules: a grammar, which defines syntactically legal terms; dynamic rules, which give the meaning of a program by defining how it is executed; and static rules, which determine whether particular programs satisfy certain well-behavedness properties. When a language has been defined we want to know its static rules are correct with respect to the dynamic rules; we outline how this can be done.

Alongside the explanation of these concepts we present `EBL`, which is a simple, typed language of arithmetic and boolean expressions. It is a language invented in this report for demonstrative purposes. Like every language we cover, it is expression-based, meaning that programs are evaluated to yield a value. `EBL` is not very mathematically interesting, but defining it gives a concrete example of the general approach thoughout this report.

2.1.1 Grammar

The grammar of a language specifies what strings are syntactically legal. A syntactically legal string is called a *term*. It is specified by giving the different categories of terms and the forms which instantiate those categories. The conventions for specifying a grammar are based on standard Backur-Naur form [1]. Figure 2.1 shows a simple grammar describing integer literals and arithmetic expressions on them. In each rule, the metavariables range over the terms of the category for which they are named.

A `EBL` program is an expression e, consisting of variable definitions, constants, and the application of boolean and arithmetic operators. A valid expression is a variable x, a constant (such as 3, 0, `true`, or `false`), the application of an operator $+$ or $\vee$ to two subexpressions, or a binding for a variable in a piece of code (`let` expression). The following are `EBL` terms: x, y, 3, $3 + 2$, $\texttt{false} \vee \texttt{true}$, $3 \vee \texttt{false}$, $\texttt{true} + \texttt{false}$, $\texttt{let}\ x = 3\ \texttt{in}\ x + 1$.

For some strings the corresponding term is ambiguous: $3 + x + 2$ could be interpreted as the term $3 + (x + 2)$ or as $(3 + x) + 2$. How we parse strings is not relevant, so we shall only consider strings which unambiguously correspond to terms. Brackets are not part of the grammar; we are just using them to disambiguate the strings we write that are meant to represent terms.

$$
\begin{array}{lll}
e & ::= & \textit{exprs}: \\
 & \mid \quad x & \textit{variable} \\
 & \mid \quad e + e & \textit{addition} \\
 & \mid \quad e \vee e & \textit{disjunction} \\
 & \mid \quad \texttt{let}\ x = e\ \texttt{in}\ e & \textit{let expr.} \\
 & & \\
v & ::= & \textit{values}: \\
 & \mid \quad l & \texttt{Nat}\ \textit{constant} \\
 & \mid \quad b & \texttt{Bool}\ \textit{constant}
\end{array}
$$

Figure 2.1: Grammar for EBL expressions.

2.1.2 Dynamic Rules

The dynamic rules of a language specify the meaning of terms. There are different approaches, but the one we use is called *small-step semantics*, where the meaning of a program is given by how it is executed. This is defined with a set of *inference rules*, which are a set of premises above a dividing line. If the premises above the line hold, they imply the *judgement* below the line. A particular application of an inference rule to obtain a judgement is a *derivation*. If an inference rule has no premises it is an *axiom*. Figure 2.2 gives the dynamic rules for EBL, which specify a binary relation $\longrightarrow$ representing a single computational step. Judgements obtained by the dynamic rules are called *reductions*. When $e \longrightarrow e'$ holds, we say that e reduces to e'.

$\boxed{e \longrightarrow e}$

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \text{(E-ADD1)} \qquad \frac{e_2 \longrightarrow e_2'}{l_1 + e_2 \longrightarrow l_1 + e_2'}\ \text{(E-ADD2)} \qquad \frac{l_1 + l_2 = l_3}{l_1 + l_2 \longrightarrow l_3}\ \text{(E-ADD3)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 \vee e_2 \longrightarrow e_1' \vee e_2}\ \text{(E-OR1)} \qquad \frac{}{\texttt{true} \vee e_2 \longrightarrow \texttt{true}}\ \text{(E-OR2)} \qquad \frac{}{\texttt{false} \vee e_2 \longrightarrow e_2}\ \text{(E-OR3)}$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2}\ \text{(E-LET1)} \qquad \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e_2 \longrightarrow [v/x]e_2}\ \text{(E-LET2)}$$

Figure 2.2: Single-step reduction rules for EBL.

An addition is reduced by first reducing the left-hand side to an irreducible form (E-ADD1) and then the right-hand side (E-ADD2). If both sides are integer literals, the expression reduces to whatever is the sum of those literals.

According to these rules, a disjunction is reduced by first reducing the left-hand side to an irreducible form (E-OR1). If the left-hand side is the boolean literal `true`, the expression reduces to `true` (because $\texttt{true} \vee Q = \texttt{true}$). Otherwise if the left-hand side is the boolean literal `false`, the expression reduces to the right-hand side e_2 (because $\texttt{false} \vee Q = Q$). This particular formulation encodes short-circuiting behaviour into $\vee$, meaning if the left-hand side is true, the whole expression will evaluate to true without checking the right-hand side.

A `let` expression is reduced by first reducing the subexpression being bound (E-LET1). If the subexpression is an irreducible form v_1, the variable x is substituted for v_1 in the body e_2 of the `let` expression (E-LET2). The notation for this is $[v_1/x]e_2$. For example, $\texttt{let } x = 1 \texttt{ in } x + 1 \longrightarrow 1 + 1$ by E-LET2.

Formally, substitution is a function on expressions. A definition is given in Figure 2.3. The notation $[e_1/x]e$ is short-hand for $\texttt{substitution}(e, e_1, x)$. For multiple substitutions we use the notation $[e_1/x_1, e_2/x_2]e$ as shorthand for $[e_2/x_2]([e_1/x_1]e)$. Note how the order of the variables has been flipped: the substitutions occur left-to-right, as they are written.

$\texttt{substitution} :: \texttt{e} \times \texttt{e} \times \texttt{v} \longrightarrow \texttt{e}$

$[e'/y]l = l$
$[e'/y]b = b$
$[e'/y]x = v$, if $x = y$
$[e'/y]x = x$, if $x \neq y$
$[e'/y](e_1 + e_2) = [e'/y]e_1 + [e'/y]e_2$
$[e'/y](e_1 \vee e_2) = [e'/y]e_1 \vee [e'/y]e_2$
$[e'/y](\texttt{let } x = e_1 \texttt{ in } e_2) = \texttt{let } x = [e'/y]e_1 \texttt{ in } [e'/y]e_2$, if $y \neq x$ and y does not occur free in e_1 or e_2

Figure 2.3: Substitution for EBL.

A robust definition of the `substitution` function is surprisingly tricky. Consider the program $\texttt{let } x = 1 \texttt{ in } (\texttt{let } x = 2 \texttt{ in } x + z)$. It contains two different variables with the same name x, with the inner one "shadowing" the outer one. Neither variable occurs "free", because both have been introduced in the body of the program (one for each `let`). Such variables are called bound variables. By contrast, z is a free variable because it has no definition in the program. A robust `substitution` should not accidentally conflate two different variables with identical names, and it should not do anything to bound variables.

To illustrate the solution, consider $\texttt{let } x = 1 \texttt{ in } (\texttt{let } x = 2 \texttt{ in } x + z)$. In some sense, this is an equivalent program to $\texttt{let } x = 1 \texttt{ in } (\texttt{let } y = 2 \texttt{ in } y + z)$, because the names of variables are arbitrary, so changing them will not change the semantics of the program. Therefore, we can freely and implicitly interchange expressions which are equivalent up to the naming of bound variables. This process is called α-conversion [20, p. 71]. Consequently, we shall assume all expressions we work with have been α-converted so all variables are uniquely named and play nice with the definition of `substitution`.

Lastly, note how in an expression like $\texttt{let } x = 1 + 1 \texttt{ in } x + 1, 1 + 1$ would first be reduced to 2 before the substitution is made on $x + 1$. This way of defining dynamic rules — so that subexpressions are reduced to irreducible forms before they are bound to their names — is the *call-by-value* strategy. Some languages — such as Haskell — are not call-by-value. In this report we only consider call-by-value semantics.

From the single-step reduction relation we define the multi-step reduction relation as a sequence of zero[1] or more single-steps. This is written $e \longrightarrow^* e'$. For example, if $e_1 \longrightarrow e_2$ and $e_2 \longrightarrow e_3$, then $e_1 \longrightarrow^* e_3$. Figure 2.4 defines multi-step reduction in EBL.

[1] We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation [20, p. 39].

$\boxed{e \longrightarrow^* e}$

$$\frac{}{e \longrightarrow^* e}\ \text{(E-MULTISTEP1)} \quad \frac{e \longrightarrow e'}{e \longrightarrow^* e'}\ \text{(E-MULTISTEP2)}$$

$$\frac{e \longrightarrow^* e' \quad e' \rightarrow^* e''}{e \longrightarrow^* e''}\ \text{(E-MULTISTEP3)}$$

Figure 2.4: Multi-step reduction rules for EBL.

E-MULTISTEP1 says that an expression can always be "reduced to itself". E-MULTISTEP2 says that a single-step reduction is also a multi-step reduction. E-MULTISTEP3 says that if a sequence of reductions can reduce e to e', and another sequence can reduce e' to e'', then $e \longrightarrow^* e''$ is a valid sequence of reductions.

2.1.3 Static Rules

When attempting to reduce EBL terms you may find you end up with nonsense, or get stuck in a situation where no rule applies due to a typing error. For example, $\texttt{false} \vee 3 \longrightarrow 3$ by E-OR3, which is nonsense. $(1 + 1) + \texttt{false} \longrightarrow 2 + \texttt{false}$ by E-ADD1, but then you are stuck because $+$ should be applied to numbers, and `false` is a boolean. Another example is $\texttt{x} + 1$, which gets stuck because x is undefined.

We often want to consider those programs which satisfy certain well-behavedness properties. One such property is that of being *well-typed*: if a program is well-typed then during execution it will never get *stuck* due to type errors, like the application of $+$ to booleans. Another is that every variable in a program must be declared before it is used. If a program satisfies these well-behavedness properties, its execution would never get stuck or produce a nonsense answer.

To determine if a program is well-behaved before it is executed we shall add static rules to EBL, enriching it with a basic type system that associates a type to well-behaved expressions. If an expression can be given a type then its execution will have no type errors. Our type system will also encode the requirement that variables be defined before they are used. The relevant constructs for the type system are given as a grammar in Figure 2.5. There are two types: `Nat` and `Bool`. A typing context Γ maps variables to their types; this is needed in a program like $\texttt{let}\ x = 1\ \texttt{in}\ x + 1$, where in typing $x + 1$, we need to know the type of x.

$$\begin{array}{lll} \tau & ::= & \textit{types}: \\ & \mid & \texttt{Nat} \\ & \mid & \texttt{Bool} \\ \\ \Gamma & ::= & \textit{contexts}: \\ & \mid & \varnothing \\ & \mid & \Gamma, x : \tau \end{array}$$

Figure 2.5: Grammar for the type system of EBL.

Figure 2.6 presents the static rules of EBL. The judgement form is $\Gamma \vdash e : \tau$, which means expression e has type τ in the context Γ. When a judgement can be derived from the empty context it is written $\vdash e : \tau$ instead of $\varnothing \vdash e : \tau$.

T-BOOL and T-NAT are rules which say that constants always type to `Bool` or `Nat`. T-

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

$$\frac{}{\Gamma, x : \texttt{Int} \vdash x : \texttt{Int}}\ \text{(T-VAR)} \quad \frac{}{\vdash b : \texttt{Bool}}\ \text{(T-BOOL)} \quad \frac{}{\vdash l : \texttt{Nat}}\ \text{(T-NAT)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \texttt{Bool}}{\Gamma \vdash e_1 \vee e_2 : \texttt{Bool}}\ \text{(T-OR)} \quad \frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 + e_2 : \texttt{Nat}}\ \text{(T-ADD)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{(T-LET)}$$

Figure 2.6: Inference rules for typing arithmetic expressions.

VAR says that a variable types to whatever the context binds it to. T-OR types a disjunction if the arguments are both `Bool`. T-ADD types a sum if the arguments are both `Nat`. The most interesting rule is T-LET, where the context gains a binding for x to type-check the body of the `let` expression. For example, $\texttt{let}\ x = 1\ \texttt{in}\ x + 1$ typechecks because $x : \texttt{Int} \vdash x + 1 : \texttt{Int}$; a derivation is given in Figure 2.7. The type of a `let` expression is the type of its body.

$$\dfrac{\dfrac{}{\vdash 1 : \texttt{Nat}}\ \text{(T-NAT)} \qquad \dfrac{\dfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}\ \text{(T-VAR)} \qquad \dfrac{}{x : \texttt{Int} \vdash 1 : \texttt{Int}}\ \text{(T-NAT)}}{x : \texttt{Int} \vdash x + 1 : \texttt{Int}}\ \text{(T-ADD)}}{\vdash \texttt{let}\ x = 1\ \texttt{in}\ x + 1 : \texttt{Int}}\ \text{(T-LET)}$$

Figure 2.7: Derivation tree for $\texttt{let}\ x = 1\ \texttt{in}\ x + 1$

There are some pesky technicalities about typing contexts which need to be addressed. Though we have defined Γ syntactically as a sequence of variable-type pairs, we really want to treat it like a set. For example, $x : \texttt{Int}, y : \texttt{Int}$ is really the same thing as $y : \texttt{Int}, x : \texttt{Int}$. By the convention of α-renaming, all programs have unique variable names, so no duplicate names will arise in practice. Some other properties we require is that whenever a judgement holds in a context Γ, it should also hold in any super-context Γ. For example, $x : \texttt{Int} \vdash x : \texttt{Int}$, but it's also true that $x : \texttt{Int}, y : \texttt{Int} \vdash x : \texttt{Int}$. The order of the bindings in Γ should not matter. We can ensure these properties with the rules in Figure 2.8.

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

$$\frac{\Gamma \vdash e : \tau \quad \Gamma' \textit{ is a permutation of } \Gamma}{\Gamma' \vdash e : \tau}\ \text{(Γ-PERMUTE)} \quad \frac{\Gamma \vdash e : \tau \quad x \notin \texttt{dom}(\Gamma)}{\Gamma, x : \tau' \vdash e : \tau}\ \text{(Γ-WIDEN)}$$

Figure 2.8: Structural rules for typing contexts.

Γ-PERMUTE says that a judgement holds in Γ if it holds in any permutation of Γ, meaning the order is irrelevant. Γ-WIDEN says that any judgement which holds in Γ will hold in $\Gamma, x : \tau$, provided x is not already in the domain of Γ. $\texttt{dom}(\Gamma)$ is the set of variables bound in Γ; a definition is given in 2.9. All of this causes typing contexts to behave as we expect, but in practice the notation for contexts and how to manipulate them is so conventional that we shall never both to mention them again. The structural rules of contexts will be automatically and implicitly applied.

Most languages have a *subtyping* judgement $\tau_1 <: \tau_2$, which means that expressions of type τ_1 are also of type τ_2. `EBL` has no subtyping rules, but we shall encounter them later.

$\texttt{dom} :: \Gamma \to \{\texttt{x}\}$

$$\texttt{dom}(\varnothing) = \varnothing$$
$$\texttt{dom}(\Gamma, x : \tau) = \texttt{dom}(\Gamma) \cup \{x\}$$

Figure 2.9: Definition of dom.

2.1.4 Soundness

If we tried to apply the static rules to the misbehaved programs from the last section, we would find there is no way to ascribe a type to them. This is good, but we want to know that every such misbehaving program is rejected by the type system. This property is called *soundness*. The exact definition depends on the language under consideration, but is often split into two parts called progress and preservation. These are given below for EBL.

Theorem 1 (EBL Preservation)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\vdash e' : \tau$*.*

Preservation states that a well-typed term is still well-typed after it has been reduced. This means a sequence of reductions will produce intermediate terms that are also well-typed and do not get stuck. In EBL, the type of the term after reduction is the same as the type of the term before reduction.

Theorem 2 (EBL Progress)**.** *If* $\vdash e : \tau$ *and* e *is not a value, then* $e \longrightarrow e'$ *for some* e'*.*

Progress states that any well-typed, non-value term can be reduced i.e. it will not get stuck due to type errors. A consequence of this is that values in the grammar are exactly the well-typed, irreducible expressions. This is intentional and we always define values to be like this. For this reason we will often refer to irreducible expressions as values, even before we have shown they are equivalent. Combining progress and preservation, we know that well-typed programs which are not-values can be single-step reduced (i.e. they are not stuck). This is single-step soundness

Theorem 3 (EBL Single-step Soundness)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\vdash e' : \tau$*, for some* e'*,* τ*.*

All these theorems are proved by structural induction on the derivation of $\Gamma \vdash e : \tau$ in the premise and then, where appropriate, on the derivation of $e \longrightarrow e'$ in the premise. Once small-step soundness has been established, multi-step soundness follows by inducting on the length of a multi-step reduction.

Theorem 4 (EBL Multi-step Soundness)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow^* e'$ *then* $\vdash e' : \tau$*, for some* e'*,* τ*.*

There are two common lemmas needed in the proof of soundness. The first is canonical forms, which outlines a set of useful observations that follow immediately from the static rules. The second is the substitution lemma, which says if a term e is well-typed in a context $\Gamma, x : \tau'$, and instances of x in e are replaced with an expression e' of type τ', then the type of e is preserved. In EBL, this lemma is needed to show that the reduction step in E-LET2 preserves types. Formulations of these lemmas are given below.

Lemma 1 (Canonical Forms)**.** *The following are true:*

- *If* $\Gamma \vdash v$: Int*, then* $v = l$ *is a* Nat *constant.*
- *If* $\Gamma \vdash v$: Bool*, then* $b = l$ *is a* Bool *constant.*

Lemma 2 (Substitution)**.** *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash [e'/x]e : \tau$*.*

Soundness establishes that the static rules reject every program violating their well-behavedness properties. The converse is also interesting to consider: if a program is well-behaved, will the type system accept it? This is called *completeness*. Most interesting type systems are incomplete. To show why, consider the Java program in Figure 2.10. This program is type-safe, because the only branch of the conditional that ever executes is the one which returns an `int`. However, Java will reject this program because the other branch returns a `boolean` determining which branches will execute is generally undecidable.

```
1 public int doubleNum(int x) {
2   if (true) return x + x;
3   else return true;
4 }
```

Figure 2.10: A type-safe Java method which does not typecheck.

This report is only concerned with proving soundness, but it is important recognise that being incomplete makes a type system inherently *conservative*, meaning it will reject type-safe programs or make static over-estimations. One view of type systems is that they "calculate a kind of static approximation to the run-time behaviours of the terms in a program" [20, p. 2]. In order to approximate, simplifying assumptions must be made, and these simplifying assumptions are what make the type-system sound and decidable; but assumptions which simplify too much will reject more and more safe programs, making the system less useful.

2.2 $\lambda^{\rightarrow}$: Simply-Typed λ-Calculus

The simply-typed λ-calculus $\lambda^{\rightarrow}$ is a model of computation, first described by Alonzo Church [5], based on the definition and application of functions. $\lambda^{\rightarrow}$ serves as the basis for many programming languages, including those in this report. We present a variant with natural numbers, integers, and subtyping. Its grammar is given in Figure 2.11.

e	::=		*exprs* :	v	::=		*values* :
	$\mid$	x	*variable*		$\mid$	$\lambda x : \tau.e$	*abstraction*
	$\mid$	$e\ e$	*application*		$\mid$	n	`Nat` *constant*
	$\mid$	v	*value*		$\mid$	i	`Int` *constant*

Figure 2.11: Grammar for $\lambda^{\rightarrow}$.

An expression in $\lambda^{\rightarrow}$ is either a variable x, the application of a function to a value $e\ e$, or a value. A value can be a `Nat` constant n, an `Int` constant i, or a function literal $\lambda x : \tau.e$ (also called an abstraction). To distinguish the strings representing `Nat` constants from the strings representing positive `Int` constants, we write $3_{\mathbb{N}}$ for the former and $3_{\mathbb{Z}}$ for the latter. This is not part of the grammar, it is just notation. A function literal $\lambda x : \tau.e$ has a function body e and a formal argument x with type τ. For example, $\lambda x : \texttt{Int}.\ x$ is the identity function on integers. $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{Z}}$ is the application of that identity function to the integer literal $3_{\mathbb{Z}}$.

A grammar for types in $\lambda^{\rightarrow}$ is given in Figure 2.12. A type context Γ is a sequence of variable bindings, interpreted in the usual way. There are two base types: `Nat` and `Int`. $\rightarrow$ ("arrow") is a type constructor: it builds a new type from existing ones. $\tau_1 \rightarrow \tau_2$ is the type of a function which takes as input a τ_1 and returns a τ_2. The function $\lambda x : \texttt{Int}.\ x$ would have the type $\texttt{Int} \rightarrow \texttt{Int}$. Some other examples of types are $\texttt{Int} \rightarrow \texttt{Nat}$, $\texttt{Nat} \rightarrow (\texttt{Int} \rightarrow \texttt{Nat})$, and $(\texttt{Nat} \rightarrow \texttt{Nat}) \rightarrow (\texttt{Nat} \rightarrow \texttt{Int})$.

$$
\begin{array}{rcll}
\Gamma & ::= & & \textit{type} : \\
& | & \texttt{Nat} & \textit{natural numbers} \\
& | & \texttt{Int} & \textit{integers} \\
& | & \tau \to \tau & \textit{arrow}
\end{array}
\qquad
\begin{array}{rcll}
\Gamma & ::= & & \textit{type ctx.} : \\
& | & \varnothing & \textit{empty ctx.} \\
& | & \Gamma, x : \tau & \textit{var. binding}
\end{array}
$$

Figure 2.12: Grammar for $\lambda^{\to}$.

Before giving the small-step semantics, we define `substitution` in Figure 2.13: numeric constants are unchanged by substitution; a variable is changed if it matches the variable being replaced; a function has the free variables in its body replaced; an application has the free variables in its subexpressions replaced.

`substitution` :: `e` × `e` × `v` → `e`

$$
\begin{array}{l}
[v/y]i = i \\
[v/y]n = n \\
[v/y]x = v, \text{ if } x = y \\
[v/y]x = x, \text{ if } x \neq y \\
[v/y](\lambda x : \tau.e) = \lambda x : \tau.[v/y]e, \text{ if } y \neq x \text{ and } y \text{ does not occur free in } e \\
[v/y](e_1\ e_2) = ([v/y]e_1)([v/y]e_2)
\end{array}
$$

Figure 2.13: Substitution for $\lambda^{\to}$.

The small-step semantics for $\lambda^{\to}$ are summarised in Figure 2.14. Multi-step semantics are defined the same as in `EBL`. The only reducible expression is a function application. E-APP1 will reduces the left-side of an application. If the left-side is a value, but the right-side is not, then E-APP2 reduces the right-side. If the left-side is a function and the right-side is a value, then E-APP3 binds the right-side to the name of the function's formal argument in the function body. For example, $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{Z}} \longrightarrow 3_{\mathbb{Z}}$ by E-APP3.

$\boxed{e \longrightarrow e}$

$$
\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1' e_2 \mid \varepsilon}\ \text{(E-APP1)} \qquad \frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1 e_2 \longrightarrow v_1 e_2' \mid \varepsilon}\ \text{(E-APP2)}
$$

$$
\frac{}{(\lambda x : \tau.e) v_2 \longrightarrow [v_2/x]e \mid \varnothing}\ \text{(E-APP3)}
$$

Figure 2.14: Single-step judgement rules for $\lambda^{\to}$.

As with `EBL`, some expressions in $\lambda^{\to}$ exhibit strange behaviours due to type errors or undefined variables. For example, consider $e = (\lambda x : \texttt{Int}.\ x)(\lambda x : \texttt{Int}.\ x)$. By E-APP3, $e \longrightarrow \lambda x : \texttt{Int}.\ x$. Intuitively we should reject this program on the left because the function on the left takes an `Int` as an argument, and the function on the right is not an `Int`. Another example is $(\lambda x : \texttt{Int}.\ y)\ 3_{\mathbb{Z}}$, which reduces to y by E-APP3 and then gets stuck. This should be rejected because y is undefined. To determine whether a program is well-behaved we can apply the static rules for $\lambda^{\to}$, summarised in Figure 2.15.

The first two rules state that a natural number constant can always be typed to `Nat` and an integer constant can always be typed to `Int`. T-VAR states that a variable bound in some context can be typed as its binding. T-ABS states that a function can be typed in Γ if Γ can type the body of the function, when the function's argument has been bound to its formal type. T-APP states that an application is well-typed if the left-hand expression reduces to a function of type $\tau_2 \to \tau_3$ and the right-hand expression has type τ_2. The examples above

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

$$\frac{}{\Gamma \vdash n : \texttt{Nat}}\ \text{(T-NAT)} \quad \frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{(T-INT)} \quad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{(T-VAR)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \text{(T-ABS)} \quad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_3}\ \text{(T-APP)}$$

Figure 2.15: Static rules for $\lambda^{\to}$.

will now reject: $(\lambda x : \texttt{Int}.\ x)\ (\lambda x : \texttt{Int}.\ x)$ does not type because $\vdash \lambda x : \texttt{Int}.\ x : \texttt{Int} \to \texttt{Int}$, but the right-hand side does not have type `Int`; $(\lambda x : \texttt{Int}.\ y)\ 3_{\mathbb{Z}}$ does not type because no rule can type y in the context $x : \texttt{Int}$.

Consider the example $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}}$, where a natural number is passed to the identity function for integers. The rules cannot type this program because the function expects an `Int`, but in some sense a `Nat` is a specific sort of `Int`, and sometimes it is convenient to treat it as such. We call `Nat` a subtype of `Int` and write $\texttt{Nat} <: \texttt{Int}$ for this judgement. In general, the judgement form $\tau_1 <: \tau_2$ means that values of type τ_1 are also values of type τ_2. We call τ_1 more specific and τ_2 less specific. Subtyping judgements for $\lambda^{\to}$ are given in Figure 2.16.

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

$$\frac{}{\tau <: \tau}\ \text{(S-REFLEXIVE)} \quad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \text{(S-TRANSITIVE)}$$

$$\frac{}{\texttt{Nat} <: \texttt{Int}}\ \text{(S-NAT)} \quad \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ \text{(S-ARROW)}$$

Figure 2.16: Static rules for $\lambda^{\to}$.

The rules S-REFLEXIVE and S-TRANSITIVE make subtyping a pre-ordering relation on types. Every system with subtyping will have these rules, so we will not write them down again. S-NAT says that natural numbers are also integers. The most intriguing rule is S-ARROW, which describes when one function is a subtype of another. Notice how the direction of the subtyping relation is flipped for the input types in the premise, whereas the direction is preserved for the output types. We say functions are *contravariant* in their input type and *covariant* in their output type.

To illustrate why S-ARROW is sensible, consider $\texttt{Int} \to \texttt{Int}$ and $\texttt{Nat} \to \texttt{Int}$. The former could take either an `Int` or a `Nat` (because $\texttt{Nat} <: \texttt{Int}$) as input, but the latter can only take a `Nat` as input. $\texttt{Int} \to \texttt{Int}$ functions can take every input a $\texttt{Nat} \to \texttt{Int}$ can, so $\texttt{Int} \to \texttt{Int} <: \texttt{Nat} \to \texttt{Int}$. The direction of this judgement is reversed from $\texttt{Nat} <: \texttt{Int}$, so input type should be contravariant. On the other hand, consider $\texttt{Int} \to \texttt{Int}$ and $\texttt{Int} \to \texttt{Nat}$. The former might return a `Nat` or an `Int`, but the latter can only return a `Nat`. It would be safe to treat $\texttt{Int} \to \texttt{Nat}$ functions as $\texttt{Int} \to \texttt{Int}$ functions, because the former only return `Nat` values, and the latter is allowed to return `Nat` values. However, $\texttt{Int} \to \texttt{Int}$ functions could return an `Int` value, so it would not be safe to treat them as a $\texttt{Int} \to \texttt{Nat}$ function, which can only return a `Nat` value. Therefore, $\texttt{Int} \to \texttt{Nat} <: \texttt{Int} \to \texttt{Int}$; the direction of this judgement is the same as $\texttt{Nat} <: \texttt{Int}$, so the output types should be covariant.

In order to typecheck an example like $\lambda x : \texttt{Int}.\ 3_{\mathbb{N}}$, we need a rule which lets us consider $3_{\mathbb{N}}$ as an `Int`. More generally, we should be able to treat any subtype as one of its supertypes. This is called subsumption; the rule for it is given in Figure 2.17. With it, the type system will now accept programs like $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}}$. A derivation for $\vdash (\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} : \texttt{Int}$ is given in Figure 2.18.

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

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \text{T-SUBSUME}$$

Figure 2.17: The subsumption rule.

$$\dfrac{\dfrac{\dfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}\ (\text{T-VAR})}{\vdash \lambda x : \texttt{Int}\ x : \texttt{Int} \to \texttt{Int}}\ (\text{T-ABS}) \qquad \dfrac{\dfrac{\dfrac{}{\vdash 3_{\mathbb{N}} : \texttt{Nat}}\ (\text{T-NAT}) \qquad \dfrac{}{\texttt{Nat} <: \texttt{Int}}\ (\text{S-NAT})}{\vdash 3_{\mathbb{N}} : \texttt{Int}}\ (\text{T-SUBSUME})}{(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} : \texttt{Int}}\ (\text{T-APP})$$

Figure 2.18: A derivation of $\vdash (\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} : \texttt{Int}$.

The definition of soundness for $\lambda^{\to}$ is similar to EBL, but in the presence of subtyping, the type can get more specific after reduction. To illustrate how this can happen, consider $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}}$. Figure 2.18 shows a derivation of $\vdash (\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} : \texttt{Int}$. By E-APP3, $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} \longrightarrow 3_{\mathbb{N}}$. Then by T-NAT, $\vdash 3_{\mathbb{N}} : \texttt{Nat}$. $\texttt{Nat} <: \texttt{Int}$, so the type got more specific. In general, if a function has input type τ then it could take any argument which is a subtype of τ. Once that argument has been reduced to a value, we can determine exactly which subtype it is.

The soundness theorem for $\lambda^{\to}$ is stated below.

Theorem 5 ($\lambda^{\to}$ Multi-step Soundness)**.** *If* $\vdash e_A : \tau_A$ *and* $e_A \longrightarrow^* e_B$*, then* $\vdash e_B : \tau_B$*, where* $\tau_B <: \tau_A$*, for some* e_B, τ_B*.*

As a short aside, $\lambda^{\to}$ (and EBL) are *Turing-incomplete*, meaning there are programs which can be written in general-purpose languages that cannot be written in $\lambda^{\to}$. There are several routine ways to make $\lambda^{\to}$ as expressive as general-purpose languages, but because this report is mainly interested in static rules, we leave our languages Turing-incomplete to simplify the formalisms and minimise irrelevant details. Being Turing-complete is essential for a general purpose programming language, but in this report, we are just demonstrating the static rules (which equally apply to Turing-incomplete program), so it is not necessary.

2.3 Effect Systems

We have seen how the static rules of a language allow us to judge whether certain well-behavedness properties hold of a piece of code in a particular typing context. Some of these well-behavedness properties include being well-typed and defining every variable before it is used. One extension to classical type systems is to incorporate a theory of *effects*. Judgements in a *type-and-effect* system associate both a type and a set of effects with well-behaved programs. Effects describe intensional information about the way in which a program executes [17]. To illustrate, we present a simplified version of Side-Effect Analysis (SEA), which is a calculus for reasoning about which memory cells are written or read during execution [17]. The small-step semantics introduce more concepts which are largely irrelevant for the rest of the report, so a brief summary is given rather than formal dynamic rules.

2.3.1 SEA: Side-Effect Analysis

SEA extends $\lambda^{\to}$ with imperative constructs by allowing memory cells to be accessed and written via references. Our goal is to determine which references might be accessed, written, or created during execution. An effect in SEA is one of these three operations on a particular

memory cell. A particular memory cell is denoted π. It can be thought of as drawn from a set of memory cell variables Π. The grammar is given in Figure 2.19. The first new form is $\texttt{new}_\pi x = e \texttt{ in } e$, which creates a new reference x pointing to cell π in the body of e_2. The contents of π are initialised with e_1. $!x$ accesses the value at the cell referenced by x. $x := e$ updates the contents of the cell referenced by x with e.

$$
\begin{array}{rcll}
e & ::= & & \textit{exprs}: \\
 & | & x & \textit{variable} \\
 & | & e\ e & \textit{application} \\
 & | & \texttt{new}_\pi x = e \texttt{ in } e & \textit{ref. creation} \\
 & | & !x & \textit{ref. access} \\
 & | & x := e & \textit{ref. update} \\
 & | & v & \textit{value}
\end{array}
\qquad
\begin{array}{rcll}
v & ::= & & \textit{values}: \\
 & | & \lambda x : \tau . e & \textit{abstraction} \\
 & | & n & \textit{natural literal} \\
 & | & b & \textit{boolean literal}
\end{array}
$$

Figure 2.19: Grammar for SEA expressions.

An effect ϕ is the creation, reading, or writing of a reference to a particular location π. For example, a program with the effect $!\pi$ is one that reads from memory cell π during execution. Creating a reference at π is $\texttt{new}_\pi$. Updating a reference at π is $\pi{:=}$. A set of effects is denoted Φ. A grammar for effects is given in Figure 2.20.

$$
\begin{array}{rcll}
\phi & ::= & & \textit{effects}: \\
 & | & \texttt{new}_\pi & \textit{ref. creation} \\
 & | & !\pi & \textit{ref. access} \\
 & | & \pi := & \textit{ref. update}
\end{array}
\qquad
\begin{array}{rcl}
\Phi & ::= & \textit{sets of effects}: \\
 & | & \{\bar{\phi}\}
\end{array}
$$

Figure 2.20: Grammar for effects and regions of SEA.

The runtime has the notion of a *store*, which maps references to the values in their cell. The store also keeps track of what location each reference points to. It can be enlarged and updated during runtime by creating, accessing, and updating references. Each of these will incur the appropriate runtime effect. Reading and writing on a reference x both evaluate to the (potentially modified) value in the cell to which x refers. Executing a program in a store yields a reduced program, the modified version of the store, and the set of effects Φ which occurred during the execution. For brevity, we omit the formal definition of these rules.

The only base type is $\texttt{Nat}$. $\tau_1 \to_\Phi \tau_2$ is the type of a function which takes a τ_1 as input and returns a τ_2 as output. The set Φ is an upper-bound on the actual effects incurred by the function: if an effect ϕ occurs at runtime, then $\phi \in \Phi$, but it is not guaranteed that every $\phi \in \Phi$ will happen. There is also a new type constructor $\texttt{ref}$. $\texttt{ref}(\tau, \pi)$ is the type of a reference that points to the cell π, which contains a value of type τ. The grammar for types is given in 2.21.

$$
\begin{array}{rcll}
\tau & ::= & & \textit{types}: \\
 & | & \texttt{Nat} & \textit{natural numbers} \\
 & | & \texttt{Bool} & \textit{booleans} \\
 & | & \tau \to \tau & \textit{functions} \\
 & | & \texttt{ref}(\tau, \pi) & \textit{references}
\end{array}
\qquad
\begin{array}{rcll}
\Gamma & ::= & & \textit{contexts}: \\
 & | & \varnothing & \textit{empty ctx.} \\
 & | & \Gamma, x : \tau & \textit{var. binding}
\end{array}
$$

Figure 2.21: Grammar for SEA types.

There is a single judgement $\Gamma \vdash e : \tau \texttt{ with } \Phi$. This can be read as meaning that, in the context Γ, e terminates yielding a value of type τ, with Φ as a conservative upper-bound on the effects incurred during execution. If $\phi \in \Phi$ it is not guaranteed to happen at runtime, but if $\phi \notin \Phi$, it cannot happen at runtime. The static rules are summarised in Figure 2.22.

$\boxed{\Gamma \vdash e : \tau \texttt{ with } \Phi}$

$$\frac{}{\Gamma \vdash b : \texttt{Bool with } \varnothing}\ \text{(T-BOOL)} \qquad \frac{}{\Gamma \vdash n : \texttt{Nat with } \varnothing}\ \text{(T-NAT)}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \texttt{ with } \varnothing}\ \text{(T-VAR)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \texttt{ with } \Phi}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to_{\Phi} \tau_2 \texttt{ with } \varnothing}\ \text{(T-ABS)}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to_{\Phi_3} \tau_3 \texttt{ with } \Phi_1 \quad \Gamma \vdash e_2 : \tau_2 \texttt{ with } \Phi_2}{\Gamma \vdash e_1\ e_2 : \tau_3 \texttt{ with } \Phi_1 \cup \Phi_2 \cup \Phi_3}\ \text{(T-APP)}$$

$$\frac{}{\Gamma, x : \texttt{ref}(\tau, \pi) \vdash !x : \tau \texttt{ with } \{!\pi\}}\ \text{(T-READ)}$$

$$\frac{\Gamma, x : \texttt{ref}(\tau, \pi) \vdash e : \tau \texttt{ with } \Phi}{\Gamma, x : \texttt{ref}(\tau, \pi) \vdash x := e : \tau \texttt{ with } \Phi \cup \{\pi :=\}}\ \text{(T-WRITE)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \texttt{ with } \Phi_1 \quad \Gamma, x : \texttt{ref}(\tau_1, \pi) \vdash e_2 : \tau_2 \texttt{ with } \Phi_2}{\Gamma \vdash \texttt{new}_{\pi} x = e_1 \texttt{ in } e_2 : \tau_2 \texttt{ with } \Phi_1 \cup \Phi_2 \cup \{\texttt{new}_{\pi}\}}\ \text{(T-NEW)}$$

Figure 2.22: Static rules for SEA.

The first two rules state that in any context, constants have their appropriate type and no effects. The next three rules are analogous to those in $\lambda^{\to}$, but with effects included. T-VAR says that any variable x has the effect $\varnothing$, so long as the context has a binding for x. T-ABS says that if the body of the function has the effects Φ, then the function types to $\tau_1 \to_{\Phi} \tau_2$. T-APP says that applying a function incurs the effects of reducing the two subexpressions to values (Φ_1 and Φ_2) and then the effects of applying the function (Φ_3).

The new rules are for references and their operations. T-READ will type $!x$ to the type τ of the value in the cell referenced by x. Its effects are $\{!\pi\}$, where π is the cell that x refers to in the context. T-WRITE also has the type τ referenced by x, but its effects are both the operation on the reference $\pi :=$ and the result of reducing the expression being assigned (Φ). T-NEW is well-typed if the initial expression e_1 of x is well-typed, and the same environment with a new binding $x : \texttt{ref}(\tau_1, \pi)$ can type the rest of the code e_2. The effects incurred by the `new` expression are those incurred by reducing the initial expresion (Φ_1) and those incurred by reducing the rest of the code (Φ_2).

The rules of SEA now give us the ability to determine which locations in memory are instantiated, modified, or accessed — and we do not have to execute the program to find out! As an example, consider the program $e = \texttt{new}_{l_1} x = 3 \texttt{ in } x := 5$, which initialises a reference at location l_1 with 3 and then updates it to 5. This can be typed as $\vdash e : \texttt{Nat with } \{l_1 :=\}$. A derivation tree is given in Figure 2.23.

Currently, the expressive power of SEA is so low that the approximations from the static rules give *exactly* the effects which will happen at runtime. In more complex languages the approximations will stop being tight upper-bounds. To show how that might happen, consider an extended version of SEA with conditional expressions. The conditional `if` e_1 `then` e_2 `else` e_3 will evaluate e_1 and check if it is `true` or `false`. If `true`, it executes e_1. If `false`, it executes e_2. A rule for conditionals is given in Figure 2.24. A conditional is well-typed if the guard e_1 types to `Bool` and the two branches type to the same τ. Its effects are

$$\dfrac{\dfrac{}{\vdash 3 : \texttt{Nat with } \varnothing}\ \text{(T-NAT)} \qquad \dfrac{\dfrac{}{x : \texttt{ref}(\texttt{Nat}, l_1) \vdash 5 : \texttt{Nat with } \varnothing}\ \text{(T-NAT)}}{x : \texttt{ref}(\texttt{Nat}, l_1) \vdash x := 5 : \texttt{Nat with } \{l_1 :=\}}\ \text{(T-WRITE)}}{\vdash \texttt{new}_{l_1} x = 3 \texttt{ in } x := 5 : \texttt{Nat with } \{l_1 :=\}}\ \text{(T-NEW)}$$

Figure 2.23: Derivation tree for $\texttt{new}_{l_1} x = 3 \texttt{ in } x := 5$.

approximated as the effects incured by reducing the guard, and the effects incurred along both branches. Only branch is executed during runtime, but in general it cannot be statically determined which branch will execute. The only safe conclusion to make is to consider both branches as having executed, with respect to the approximated effects.

$\boxed{\Gamma \vdash e : \tau \texttt{ with } \Phi}$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool with } \Phi_1 \quad \Gamma \vdash e_2 : \tau \texttt{ with } \Phi_2 \quad \Gamma \vdash e_3 : \tau \texttt{ with } \Phi_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau \texttt{ with } \Phi_1 \cup \Phi_2 \cup \Phi_3}\ \text{(T-COND)}$$

Figure 2.24: Static rules for SEA.

2.4 The Capability Model

A *capability* is a unique, unforgeable reference, granting its bearer permission to perform some operation [7]. If a piece of code possesses a capability C, it is said to have *authority* over it. In the capability model, authority can only proliferate in the following ways [16]:

1. By the initial set of capabilities passed into the program (initial conditions).
2. If a function or object is instantiated by its parent, the parent gains a capability for its child (parenthood).
3. If a function or object is instantiated by a parent, the parent may endow its child with any capabilities it possesses (endowment).
4. A capability may be transferred via method-calls or function applications (introduction).

The proliferation rules are summarised as: "only connectivity begets connectivity." Any authority in a program either exists from the beginning from those initial capabilities passed in by the system environment, or derive from previous access. A primitive capability grants operations over *resources* in the system environment. For example, a `File` might grant operations on a particular file in the file system. We will often conflate primitive capabilities with the system resources they grant operations upon, referring to both as resources. An example of a non-primitive capability might be a `Logger` which exercises authority over a `File` and presents a `log` function which appends to it. If an effect is interpreted as some operation performed upon a system resource, then capabilities encapsulate the source of effects; the only way to incur an effect is to possess a capability for it.

If a component uses a capability which it has not been explicitly given, it is exercising *ambient authority*. Figure 2.25 demonstrates ambient authority in Java: a malicious implementation of `List.add` attempts to overwrite the user's `.bashrc` file. `MyList` gains a capability for this operation by importing `java.io.File` and instantiating new instances of a capability for the user's `.bashrc` file. Nobody has explicitly given it a capability for the `.bashrc` file; it

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

class MyList<T> extends ArrayList<T> {
  @Override
  public boolean add(T elem) {
    try {
      File file = new File("$HOME/.bashrc");
      file.createNewFile();
    } catch (IOException e) {}
    return super.add(elem);
  }
}
```

```
import java.util.List;

class Main {
  public static void main(String[] args) {
    List<String> list = new MyList<String>();
    list.add(``doIt'');
  }
}
```

Figure 2.25: `Main` exercises ambient authority over a `File` capability.

simply creating one for itself. Another way to exercise ambient authority is through global state: if a capability is stored inside a global variable then any component can use it, regardless of whether it had been given to them. Ambient authority is a challenge to the principle of least authority because it makes it impossible to determine from a module's signature what authority is being exercised. From the perspective of `Main`, knowing that `MyList.add` has a capability for the user's `.bashrc` file requires one to inspect the source code of `.bashrc`. In a large code-base, this is tedious and error-prone. If the source code is obfuscated, it is also frustrating. If the developer does not have access to the source code, it is impossible! A language in which authority is explicit and only proliferates according to the capability model is called *capability-safe*. To be capability-safe, a language must not allow unrestricted imports or global state. The result is that every component must select and be instantiated with the authority it needs to function. The implementation of `MyList` has been rewritten in Figure **??** so it exercises authority over `File` in an explicit manner.

Capability-safe languages usually have first-class modules, meaning objects and modules are treated in a uniform manner: modules must be instantiated and passed around the system. Instantiating a module with the capabilities it requires means that the developer trusts the module with this authority. Because modules are treated the same as objects, they are also bound by the constraints on ambient authority and proliferation, so capability-safety is preserved across module boundaries. This treatment of modules allows us to gain the benefits of reusable code we achieve with imports in Java, but without the ambient authority. First-class modules are not exclusive to capability-safe languages: Scala has first class modules [19], but is not capability-safe. Within the capability-safe languages there is a variation in style and design: Newspeak is dynamically-typed capability-safe language with first-class modules [3] and Wyvern is a statically-typed capability-safe language [18] with first-class modules [11].

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.ArrayList;
4
5  class MyList<T> extends ArrayList<T> {
6
7    private File f;
8
9    public MyList(File f) {
10     this.f = f;
11   }
12
13   @Override
14   public boolean add(T elem) {
15     try {
16       f.createNewFile();
17     } catch (IOException e) {}
18     return super.add(elem);
19   }
20
21 }
```

Figure 2.26: `MyList` now exercises explicit authority over the `File`.

Chapter 3

Effect Calculi

This chapter introduces a pair of languages: the operation calculus OC and the capability calculus CC. OC is an extension of $\lambda^{\rightarrow}$ with a notion of primitive capabilities and their operations. Every function is annotated with the effects it may incur and the static rules of OC ascribe a type and a set of effects to programs; the resulting theory is sound with respect to both types and effects. We then generalise OC to obtain CC, which allows unannotated code to be nested inside annotated code using a new `import` construct in a capability-safe manner. A safe inference can be made about what effects the unannotated code might incur by inspecting the capabilities it is given.

The motivating examples in this chapter are written in a *Wyvern*-like language. Wyvern is a capability-safe, pure, object-oriented language with first-class modules [18, 11]. A more thorough discussion of Wyvern and how its programs can be translated into the calculi is given in Chapter 4.

3.1 OC: Operation Calculus

.

Primitive capabilities are given to a program by the system environment and allow their bearer to perform some operations on a particular resource in the system environment. For example, a `File` might provide read/write operations on a particular file in the file system. For convenience, we often conflate primitive capabilities with their resources. An effect in OC is a particular operation invoked on some resource; for example, `File.write`. The pieces of an OC program are (conservatively) annotated with the effects they may incur during execution of the function body. Annotations might be given in accordance with the principle of least authority, to denote the maximum authority a particular component or function is allowed to exercise. When this authority is being exceeded, an effect system like that in OC will reject the program, signalling that it is unsafe. For example, consider the pair of modules in Figure 3.1: the `Logger` possesses a `File` capability and exposes a single function `log`. The `Client` has a single function `run` which, when passed a `Logger`, will invoke `Logger.log` function.

`Client.run` and `Logger.log` are both annotated with `File.append`, but the (potentially malicious) implementation of `Logger` logger incurs `File.read`. By the end of this section, we will have developed rules for OC that can approximate what effects a piece of code will incur when executed, and whether the implementation of a component is consistent with its specification. The definition of soundness is also refined to consider whether the static rules safely approximate the runtime effects.

```
resource module Logger
require File

def log(): Unit with {File.append} =
   File.read
```

```
module Client

def run(l: Logger): Unit with {File.append} =
   l.log()
```

Figure 3.1: The implementation of `Logger.log` exceeds its specified authority.

3.1.1 OC Grammar

In addition to the forms from $\lambda^{\to}$, OC contains two new forms: resource literals and operation calls. The grammar for OC programs is given in Figure 3.2.

$$
\begin{array}{rcrl}
e & ::= & & \textit{exprs}: \\
& | & x & \textit{variable} \\
& | & v & \textit{value} \\
& | & e\ e & \textit{application} \\
& | & e.\pi & \textit{operation call}
\end{array}
\qquad
\begin{array}{rcrl}
v & ::= & & \textit{values}: \\
& | & r & \textit{resource literal} \\
& | & \lambda x : \tau.e & \textit{abstraction}
\end{array}
$$

Figure 3.2: Grammar for OC programs.

A resource literal r is a variable drawn from a fixed set R. Resources cannot be created or destroyed at runtime. They model the primitive capabilities passed into the program. `File` and a `Socket` are examples of resource literals. An operation call $e.\pi$ is the invocation of an operation π on e. For example, invoking the `open` operation on `File` would be `File.open`. Operations are drawn from a fixed set Π. Neither resources nor operations can be created or destroyed at runtime, they simply exist for the entire duration of the program.

An effect is a pair $(r, \pi) \in R \times \Pi$. Sets of effects are denoted ε; a syntactic definition is given in Figure 3.3. As a shorthand, we write $r.\pi$ instead of (r, π). Effects should be distinguished from operation calls: an operation call is the invocation of a particular operation on a particular resource in a program, while an effect is a mathematical object describing this behaviour. The notation $r.*$ is short-hand for the set $\{r.\pi \mid \pi \in \Pi\}$, which contains every effect on r. Sometimes we abuse notation by conflating the effect $r.\pi$ with a singleton set of effects $\{r.\pi\}$. We might also write things like $\{r_1.*, r_2.*\}$, which should be understood as the set of all operations on r_1 and r_2.

$$
\begin{array}{rcrl}
\varepsilon & ::= & & \textit{effect sets}: \\
& | & \{\overline{r.\pi}\} & \textit{effect set}
\end{array}
$$

Figure 3.3: Grammar for effect sets in OC.

OC does not model the semantics of particular operations. In practice, operations might take arguments or return values: `File.write` would be given the value to be written and `System.random` would return a number randomly-generated by the system. However, the rules of OC are only concerned with tracking which effects occur where, so as a simplifying assumption all operations are null-ary, defined on every resource, and return a dummy value.

3.1.2 OC Dynamic Rules

Before giving the dynamic rules we extend the definition of `substitution` to be defined on the new forms in OC. The extra cases are given in Figure 3.4. We also make an extra restriction that a variable can only be substituted for a value. This restriction is imposed because if a variable can be replaced with an arbitrary expression, we might also be introducing arbitrary effects, violating the type-and-effect safety of the static rules. We only consider the call-by-value strategy, so expressions are reduced to values before being bound to names and the only substitutions during a reduction are for values, and this restriction is no problem.

`substitution` :: `e` × `v` × `v` → `e`

$$[v/y]r = r$$
$$[v/y](e_1.\pi) = ([v/y]e_1).\pi$$

Figure 3.4: Extra cases for `substitution` in OC.

pair

During reduction an operation call may be evaluated. When this happens a runtime effect is said to have taken place. Reflecting this, the form of the single-step reduction judgement is now $e \longrightarrow e \mid \varepsilon$, which means that e reduces to e', incurring the set of effects ε in the process. In the case of single-step reduction, ε is at most a single effect. Judgements for single-step reductions are given in Figure 3.5.

$\boxed{e \longrightarrow e \mid \varepsilon}$

$$\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1' \; e_2 \mid \varepsilon} \text{ (E-App1)} \qquad \frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1 \; e_2 \longrightarrow v_1 \; e_2' \mid \varepsilon} \text{ (E-App2)} \qquad \frac{}{(\lambda x : \tau.e)v_2 \longrightarrow [v_2/x]e \mid \varnothing} \text{ (E-App3)}$$

$$\frac{e \to e' \mid \varepsilon}{e.\pi \longrightarrow e'.\pi \mid \varepsilon} \text{ (E-OperCall1)} \qquad \frac{}{r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}} \text{ (E-OperCall2)}$$

Figure 3.5: Single-step reductions in OC.

The first three rules are analogous to the rules from $\lambda^{\to}$. E-App1 and E-App2 incur the effects of reducing their subexpressions. Because E-App3 is simply performing a substitution, it incurs no effects. The first new rule is E-OperCall1, which reduces the receiver of an operation call; the effects incurred are the effects incurred by reducing the receiver. When an operation π is invoked on a resource literal r, E-OperCall2 will reduce it to `unit`, incurring $\{r.\pi\}$ as a result. For example, `File.write` $\longrightarrow$ `unit` $\mid$ `File.write`$\}$ by E-OperCall2. `unit` is a derived form which has the property of being the only value of its type, which is also called `Unit`; because of this, it is used to represent the absence of information. OC does not model the semantics of operation calls, so `unit` is a sensible dummy value for operation calls to be returning. A formal treatment of `unit` and its properties is given in subsection 4.1.1.

Multi-step reductions can be defined over single-step reductions; this is given in Figure 3.6. A multi-step reduction consists of zero or more single-steps, and the resulting effect-set is the union of all the effect-sets produced by the intermediate single-steps.

s

$$\boxed{e \longrightarrow^* e \mid \varepsilon}$$

$$\frac{}{e \to^* e \mid \varnothing}\ \text{(E-MULTISTEP1)} \qquad \frac{e \to e' \mid \varepsilon}{e \to^* e' \mid \varepsilon}\ \text{(E-MULTISTEP2)}$$

$$\frac{e \to^* e' \mid \varepsilon_1 \quad e' \to^* e'' \mid \varepsilon_2}{e \to^* e'' \mid \varepsilon_1 \cup \varepsilon_2}\ \text{(E-MULTISTEP3)}$$

Figure 3.6: Multi-step reductions in OC.

3.1.3 OC Static Rules

The base types of OC are sets of resources, which are denoted $\{\bar{r}\}$. If an expression e has type $\{\bar{r}\}$, it should be interpreted as meaning that e will reduce to one of the literals in $\bar{r}$ (assuming e terminates). There is a single type-constructor $\to_\varepsilon$. $\varnothing$ is a valid type, but as there is no empty resource literal, no expression has this type. $\tau_1 \to_\varepsilon \tau_2$ is the type of a function which takes a τ_1 as input, returns a τ_2 as output, and whose body incurs no more than those effects in ε. ε is a conservative bound: if an effect $r.\pi \in \varepsilon$, then it is not guaranteed that $r.\pi$ will occur during function execution, but if $r.\pi \notin \varepsilon$ it cannot occur during function execution. A grammar for types is given in Figure 3.7. Though resource sets are defined syntactically as a sequence, they should be intrepreted as sets: $\{\texttt{File}, \texttt{Socket}\}$ and $\{\texttt{Socket}, \texttt{File}\}$ are the same type.

$$\begin{array}{rcll} \tau & ::= & & \textit{types}: \\ & \mid & \{\bar{r}\} & \\ & \mid & \tau \to_\varepsilon \tau & \end{array} \qquad \begin{array}{rcll} \Gamma & ::= & & \textit{type ctx}: \\ & \mid & \varnothing & \\ & \mid & \Gamma, x : \tau & \end{array}$$

Figure 3.7: Grammar for types in OC.

To illustrate some function types, if $\texttt{log}_1$ has the type $\{\texttt{File}\} \to_{\texttt{File.append}} \texttt{Unit}$, then invoking $\texttt{log}_1$ will either incur File.append or no effects. If $\texttt{log}_2$ has the type $\{\texttt{File}\} \to_{\texttt{File.}*} \texttt{Unit}$, then invoking $\texttt{log}_2$ might incur any effect on File.

Given a program, we want to know what effects might be incurred when it is executed. For example, $(\lambda c : \{\texttt{File}, \texttt{Socket}\}.\ \texttt{c.write})\ \texttt{File}$ incurs File.write when executed. In deciding whether to trust a piece of code, knowing what effects it can incur might signal to us that it is unsafe or malicious. For example, consider $\texttt{log}_3 = \lambda f : \texttt{File}.\ e$, which is a logging function that takes a Socket and a File as an argument and then executes e. Suppose this function were to typecheck as $\{\texttt{File}\} \to_{\texttt{File.}*} \texttt{Unit}$ — seeing that invoking this function could incur any effect on File, a developer may therefore choose not to trust this particular logging function and will not execute it.

In this spirit, the static rules of OC associate well-typed programs with a type and a set of effects. The judgement form is $\Gamma \vdash e : \tau\ \texttt{with}\ \varepsilon$, which can be interpreted as meaning that e will reduce to a term of type τ (assuming it terminates) and the reductions will incur no more effects than those in ε. Static rules for OC are given in Figure 3.9.

ε-VAR approximates the runtime effects of a variable as $\varnothing$. ε-RESOURCE does the same for resource literal. When typing a piece of code, the typing context should always contain bindings for those resources being used in the program. For example, static judgements about File.write are only sensible if it typechecks in the environment $\texttt{File} : \{\texttt{File}\}$. Although a resource captures several effects (namely, every possible operation on itself), attempting to "reduce" a resource will incur no effects. For a similar reason, ε-ABS approx-

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \texttt{ with } \varnothing}\ (\varepsilon\text{-VAR}) \qquad \frac{}{\Gamma, r : \{r\} \vdash r : \{r\} \texttt{ with } \varnothing}\ (\varepsilon\text{-RESOURCE})$$

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau_3 \texttt{ with } \varepsilon_3}{\Gamma \vdash \lambda x : \tau_2 . e : \tau_2 \to_{\varepsilon_3} \tau_3 \texttt{ with } \varnothing}\ (\varepsilon\text{-ABS}) \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to_{\varepsilon} \tau_3 \texttt{ with } \varepsilon_1 \quad \Gamma \vdash e_2 : \tau_2 \texttt{ with } \varepsilon_2}{\Gamma \vdash e_1\ e_2 : \tau_3 \texttt{ with } \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon}\ (\varepsilon\text{-APP})$$

$$\frac{\Gamma \vdash e : \{\bar{r}\} \quad \forall r \in \bar{r} \mid r : \{r\} \in \Gamma \quad \pi \in \Pi}{\Gamma \vdash e.\pi : \texttt{Unit with } \{\bar{r}.\pi\}}\ (\varepsilon\text{-OPERCALL})$$

$$\frac{\Gamma \vdash e : \tau \texttt{ with } \varepsilon \quad \tau <: \tau' \quad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash e : \tau' \texttt{ with } \varepsilon'}\ (\varepsilon\text{-SUBSUME})$$

Figure 3.8: Type-with-effect judgements in OC.

imates the effects of a function literal as $\varnothing$. However, the ascribed type has an arrow with a set of effects, which are those the function might incur if executed. ε-APP approximates a lambda application as incurring those effects from evaluating the subexpressions and the effects incurred by executing the body of the function to which the left-hand side evaluates. The effects of a function body are approximated from its arrow-type.

Because an operation call on a resource literal will reduce to `unit`, ε-OPERCALL ascribes the type of an operation call as `Unit`. The approximate effects of an operation call are: the effects of reducing the subexpression, and then the operation π on every possible resource which that subexpression to which that subexpression might reduce. For example, consider $e.\pi$, where $\Gamma \vdash e : \{\texttt{File}, \texttt{Socket}\}$ `with` $\varnothing$. Then e could evaluate to `File`, in which case the actual runtime effect is `File`.π, or it could evaluate to `Socket`, in which case the actual runtime effect is `Socket`.ß. Determining which will happen is, in general, undecidable. The safe approximation then is to treat them both as happening. The type of an operation call is `Unit`, which is the type of `unit`. `Unit` is also a derived type, and $\vdash$ `unit` : `Unit with` $\varnothing$ by a derived rule ε-UNIT. Definitions for this are given in Chapter 4.

The last rule, ε-SUBSUME, only makes sense in the presence of subtyping. It says that the type can be narrowed or the effect-set widened on an existing judgement to produce a new judgement. The subtyping judgement are given in Figure 3.9.

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

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2' \quad \varepsilon \subseteq \varepsilon'}{\tau_1 \to_{\varepsilon} \tau_2 <: \tau_1' \to_{\varepsilon'} \tau_2'}\ (\text{S-ARROW}) \qquad \frac{r \in r_1 \implies r \in r_2}{\{\bar{r}_1\} <: \{\bar{r}_2\}}\ (\text{S-RESOURCE})$$

Figure 3.9: Subtyping judgements of OC.

The first subtyping rule is S-ARROW, which is similar to the rule for subtyping functions in $\lambda^{\to}$, but with an extra premise that the effects on the arrow of the subtype must be contained in the effects on the arrow of the supertype. To illustrate why, consider a `log` function which is allowed to open, close, and append to files. Its interface might have the type `Unit` $\to_{\{\texttt{File.open}, \texttt{File.close}, \texttt{File.append}\}}$ `Unit`. However, an implementation of `log` might open `File` and append to it, but not close it. In this case it is still meets the authority stipulated by the interface, and should be considered as a subtype. The other subtyping rule

is S-RESOURCE, which says a subset of resources is also a subtype. To justify this rule, consider $\{\bar{r}_1\} <: \{\bar{r}_2\}$. Any value with type $\{\bar{r}_1\}$ can reduce to any resource literal in $\bar{r}_1$, so to be compatible with an interface $\{\bar{r}_2\}$, the resource literals in $\bar{r}_1$ must also be in $\bar{r}_2$.

These rules let us determine what sort of effects might be incurred when a piece of code is executed. For example, consider $c = (\lambda f : \{\texttt{File}, \texttt{Socket}\}.\ \texttt{f.write})\ \texttt{File}$. The judgement $\vdash c : \texttt{Unit with } \{\texttt{File.write}, \texttt{Socket.write}\}$ holds, which means executing e might incur `File.write` or `Socket.write`. A derivation for it is given in Figure 3.10. To fit in one diagram, all resources and operations have been abbreviated to their first letter. A developer who only expects a file-writer to be incurring `File.write` can typecheck `fwrite`, see that it could also be writing to `Socket`, and decide it should not be used. If the client code c were to annotate its effects as $\{\texttt{File.write}\}$, the type system would reject c as an implementation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \{\texttt{F},\texttt{S}\} \vdash f : \{\texttt{F},\texttt{S}\}}\ (\varepsilon\text{-VAR})}{\texttt{f} : \{\texttt{F},\texttt{S}\} \vdash \texttt{f.w} : \texttt{Unit with } \{\texttt{F.w},\texttt{S.w}\}}\ (\varepsilon\text{-OPERCALL})}{\lambda \texttt{f} : \{\texttt{F},\texttt{S}\}.\texttt{f.w} : \{\texttt{F},\texttt{S}\} \to_{\texttt{F.w},\texttt{S.w}} \texttt{Unit with } \varnothing}\ (\varepsilon\text{-ABS}) \qquad \dfrac{}{\vdash \texttt{F} : \{\texttt{F}\} \texttt{ with } \varnothing}\ (\varepsilon\text{-RESOURCE})}{\vdash (\lambda \texttt{f} : \{\texttt{F},\texttt{S}\}.\texttt{f.write})\ \texttt{F} : \texttt{Unit with } \{\texttt{F.w},\texttt{S.w}\}}\ (\varepsilon\text{-APP})$$

Figure 3.10: Derivation tree for $\vdash c : \texttt{Unit with } \{\texttt{File.write}, \texttt{Socket.write}\}$.

3.1.4 OC Soundness

To show the rules of OC are sound requires an appropriate notion of the static approximations being correct with respect to the reductions. If a static judgement like $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$ were correct, then successive reductions on e should never produce effects not in ε. Adding this to our definition of soundness yields the following first definition.

Theorem 6 (OC Soundness 1)**.** *If $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ and e_A is not a value, then $e_A \longrightarrow e_B \mid \varepsilon$, where $\Gamma \vdash e_B : \tau_B \texttt{ with } \varepsilon_B$ and $\tau_B <: \tau_A$ and $\varepsilon \subseteq \varepsilon_A$, for some $e_B, \varepsilon, \tau_B, \varepsilon_B$.*

We assume that Γ is some context which contains bindings for all the resources in e_A. In the theorem statement, ε_A is an approximation to what e_A will do when executed. e_A reduces to e_B, incurring the effects in ε. e_B can be typed in the same context Γ with type τ_B and approximate effects ε_B. τ_B must be a subtype of τ_A and the runtime effects ε must be contained in the original approximation ε_A, but nothing about ε_B is stipulated. Our proof that multi-step reduction is sound will inductively appeal to single-step soundness. This is hard to do with the given definition, because it only relates the runtime effects to the approximation of the runtime effects before reduction. To accommodate a proof of multi-step soundness, we need a stronger version of soundness which explains how the approximate effects after reduction (ε_B) relate to the approximate effects before reduction (ε_A).

First consider how the type after reduction relates to the type before reduction. In λ-calculi, the type after reduction is either the same or more specific than the type before reduction (i.e. $\tau_B <: \tau_A$). The idea is that as we reduce the expression we gain more information about what it is, and can give a more precise type for it. Similarly, we want to allow for the approximation to get more specific after a reduction. To illustrate how this can happen, consider the function $\texttt{get} = \lambda x : \{\texttt{File}, \texttt{Socket}\}.x$ and the program $(\texttt{get File}).\texttt{write}$. In the context $\Gamma = \texttt{File} : \{\texttt{File}\}$, the rule ε-APP can be used to approximate the effects of $(\texttt{f File}).\texttt{write}$ as $\{\texttt{File.write}, \texttt{Socket.write}\}$. By E-APP3 we have the reduction $(\texttt{get File}).\texttt{write} \longrightarrow \texttt{File.write} \mid \varnothing$. The same context can use ε-OPERCALL to

approximate the reduced expression `File.write` as $\{$`File.write`$\}$ — note how the approximation of effects is more precise after reduction. This example shows why the approximation after reduction (ε_B) should be a subset of the approximation before reduction (ε_A). By adding this premise we have our final definition of soundness.

Theorem 7 (OC Single-step Soundness)**.** *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* e_A *is not a value, then* $e_A \longrightarrow e_B \mid \varepsilon$*, where* $\Gamma \vdash e_B : \tau_B$ `with` ε_B *and* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $e, \varepsilon, \tau_B, \varepsilon_B$*.*

Our approach to proving soundness is to show progress and preservation, which in turn rely on canonical forms and the substitution lemma, modified for OC. Canonical forms is given below. The results are not true if the rule used is ε-SUBSUME (because the type and approximate effects of a value can be arbitrarily widened), so we must exclude that.

Lemma 3 (OC Canonical Forms)**.** *Unless the typing rule used was* ε-SUBSUME*, the following are true:*

1. *If* $\Gamma \vdash x : \tau$ `with` ε *then* $\varepsilon = \varnothing$*.*
2. *If* $\Gamma \vdash v : \tau$ `with` ε *then* $\varepsilon = \varnothing$*.*
3. *If* $\Gamma \vdash v : \{\bar{r}\}$ `with` ε *then* $v = r$ *and* $\{\bar{r}\} = \{r\}$*.*
4. *If* $\Gamma \vdash v : \tau_1 \to_{\varepsilon'} \tau_2$ `with` ε *then* $v = \lambda x : \tau.e$*.*

The first two observations state that variables and values will always type with $\varnothing$ as their approximate effects. The third states that if a value is typed to a set of resources, the set of a singleton $\{r\}$ and the value is the resource literal r. The fourth states that if a value types to a function then it is a function literal. Progress follows from Canonical Forms.

Theorem 8 (OC Progress)**.** *If* $\Gamma \vdash e : \tau$ `with` ε *and* e *is not a value or variable, then* $e \longrightarrow e' \mid \varepsilon$*, for some* e', ε*.*

Proof. By induction on $\Gamma \vdash e : \tau$ `with` ε, for e not a value. If the rule is ε-SUBSUMPTION it follows by inductive hypothesis. If e has a reducible subexpression then reduce it. Otherwise use one of ε-APP3 or ε-OPERCALL2. □

To show preservation holds we need to know that type-and-effect safety, as it has been formulated in the definition of soundness, is preserved by the substitution in E-APP3. As noted in the definition of `substitution`, variables can only be substituted for values in OC. Canonical Forms tells us that any value will have its effects approximated as $\varnothing$ (unless ε-Subsume is used). Beyond this observation, the proof is routine.

Lemma 4 (OC Substitution)**.** *If* $\Gamma, x : \tau' \vdash e : \tau$ `with` ε *and* $\Gamma \vdash v : \tau'$ `with` $\varnothing$ *then* $\Gamma \vdash [v/x]e : \tau$ `with` ε*.*

Proof. By induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$ `with` ε. □

With this lemma, we can prove the preservation theorem.

Theorem 9 (OC Preservation)**.** *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* $e_A \longrightarrow e_B \mid \varepsilon$*, then* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $e_B, \varepsilon, \tau_B, \varepsilon_B$*.*

Proof. By induction on the derivation of $\Gamma \vdash e_A : \tau_A$ `with` ε_A, and then the derivation of $e_A \longrightarrow e_B \mid \varepsilon$. Since e_A can be reduced, we need only consider those rules which apply to non-values and non-variables.

Case: ε-APP Then $e_A = e_1\ e_2$ and $e_1 : \tau_2 \to_\varepsilon \tau_3$ `with` ε_1 and $\Gamma \vdash e_2 : \tau_2$ `with` ε_2. If the reduction rule used was E-APP1 or E-APP2, then the result follows by applying the inductive hypothesis to e_1 and e_2 respectively. Otherwise the rule used was E-APP3. Then

$(\lambda x : \tau_2.e)v_2 \longrightarrow [v_2/x]e \mid \varnothing$. By inversion on the typing rule for $\lambda x : \tau_2.e$ we know $\Gamma, x : \tau_2 \vdash e : \tau_3$ `with` ε_3. By canonical forms, $\varepsilon_2 = \varnothing$ because $e_2 = v_2$ is a value. Then by the substitution lemma, $\Gamma \vdash [v_2/x]e : \tau_3$ `with` ε_3. By canonical forms, $\varepsilon_1 = \varepsilon_2 = \varnothing = \varepsilon_C$. Therefore $\varepsilon_A = \varepsilon_3 = \varepsilon_B \cup \varepsilon_C$.

Case: ε-OPERCALL. Then $e_A = e_1.\pi$ and $\Gamma \vdash e_1 : \{\bar{r}\}$ `with` ε_1. If the reduction rule used was E-OPERCALL1 then the result follows by applying the inductive hypothesis to e_1. Otherwise the reduction rule used was E-OPERCALL2 and $v_1.\pi \longrightarrow$ `unit` $\mid \{r.\pi\}$. By assumption, $\Gamma \vdash v_1.\pi$: `unit with` $\{r.\pi\}$, and by ε-UNIT, $\Gamma \vdash$ `unit` : `Unit with` $\varnothing$. Therefore, $\tau_B = \tau_A =$ `Unit` and $\varepsilon \cup \varepsilon_B = \{r.\pi\} = \varepsilon_A$.

□

Our single-step soundness theorem now holds immediately by joining the progress and preservation theorems into one.

Theorem 10 (OC Single-step Soundness)**.** *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* e_A *is not a value, then* $e_A \longrightarrow e_B \mid \varepsilon$*, where* $\Gamma \vdash e_B : \tau_B$ `with` ε_B *and* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $e_B, \varepsilon, \tau_B, \varepsilon_B$*.*

Proof. If e_A is not a value then the reduction exists by the progress theorem. The rest follows by the preservation theorem. □

Knowing that single-step reductions are sound, the soundness of multi-step reductions can be shown by inductively applying single-step soundness on their length.

Theorem 11 (OC Multi-step Soundness)**.** *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* $e_A \longrightarrow^* e_B \mid \varepsilon$*, where* $\Gamma \vdash e_B : \tau_B$ `with` ε_B *and* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*.*

Proof. By induction on the length of the multi-step reduction. If the length is 0 then $e_A = e_B$ and the result holds vacuously. If the length is $n + 1$, then the first n-step reduction is sound by inductive hypothesis and the last step is sound by single-step soundness, so the entire $n + 1$-step reduction is sound. □

3.2 CC: Capability Calculus

OC requires every function to be annotated. The verbosity of such effect systems has been given as a reason for why they have not seen widespread use [21] — if we relax the requirement that all code be annotated, can a type system say anything useful about the parts which are not? The structured mixing of annotated and unannotated code can alleviate the problem by allowing developers to rapidly prototype in the unannotated sublanguage and incrementally add annotations as they are needed, giving a balance between convenience and safety. However, reasoning about unannotated code is difficult in general. Figure 3.11 demonstrates the issue: `someMethod` takes a function f as input and executes it, but the effects of f depend on its implementation. Without more information, such as extra constraints on the problem, more annotations, or a more complex type system, there is no way to know what effects might be incurred by `someMethod`.

```
1 def someMethod(f: Unit → Unit):
2   f()
```

Figure 3.11: What effects can `someMethod` incur?

A capability-safe design can help us: because the only authority which code can exercise is that which is explicitly given to them, the only capabilities that the unannotated code can use must be passed into it. If these capabilities are being passed in from an annotated environment, then we know what effects they capture. These effects are therefore a conservative upper bound on what can happen in the unannotated code. To demonstrate, consider a developer who wants to decide whether to use the `Logger` module in Figure 3.12. It must be instantiated with two capabilities, `File` and `Socket`, and provides an unannotated function `log`.

```
1 resource module Logger
2 require File
3 require Socket
4
5 def log(x: Unit): Unit
6   ...
```

Figure 3.12: In a capability-safe setting, `Logger` can only exercise authority over the `File` and `Socket` capabilities given to it.

What effects will be incurred if `Logger.log` is invoked? One approach is to manually examine its source code, but this is tedious and error-prone. In many real-world situations the source code may be obfuscated or unavailable. A capability-based argument can do better: the only authority which `Logger` can exercise is that which it has been explicitly given. Here, the `Logger` can be given a `File` and a `Socket`, so $\{\texttt{File}.*, \texttt{Socket}.*\}$ is an upper bound on the effects of `Logger`. Knowing `Logger` could be manipulating sockets, the developer decides this implementation cannot be trusted does not use it.

The reasoning we employed only required us to examine the interface of the unannotated code for the capabilities that must be passed into it. To model this situation in `CC`, we add a new `import` expression that selects what authority ε the unannotated code may exercise. In the above case, `Logger` would be selecting $\{\texttt{File}.*, \texttt{Socket}.*\}$. The static rules can then check if the capabilities being passed in by the surrounding annotated environment violate ε. If not, then ε is a safe approximation of the effects of the unannotated code. This is the key result of this report: when unannotated code is nested inside annotated code, capability-safety enables us to make a safe inference about its effects by examining what capabilities are being passed in by the annotated code.

3.2.1 `CC` Grammar

The grammar of `CC` is split into rules for annotated code and analogous rules for unannotated code. To distinguish the two, we put a hat above annotated types, expressions, and contexts: $\hat{e}$, $\hat{\tau}$, and $\hat{\Gamma}$ are annotated, while e, τ, and Γ are unannotated. The rules for unannotated programs and their types are given in Figure 3.13.

The rules are much the same as in `OC`, but the sole type-constructor $\to$ is not annotated with a set of effects. If an expression e is associated with the type $\tau_1 \to \tau_2$, it means e is a function which, when given a τ_1, will return a τ_2. This type says nothing about what effects may or may not be incurred by e. Unannotated types τ are built using $\to$ and sets of resources $\{\bar{r}\}$. An unannotated context Γ maps variables to unannotated types. Rules for annotated programs and their types are given in Figure 3.14. Except for the new `import` expression, the rules are the same as in `OC`. Annotated types $\hat{\tau}$ are built using the type constructor $\to_\varepsilon$ and sets of resources $\{\bar{r}\}$. An annotated context $\hat{\Gamma}$ maps variables to annotated types.

$$
\begin{array}{lcll}
e & ::= & & \textit{exprs}: \\
 & | & x & \textit{variable} \\
 & | & v & \textit{value} \\
 & | & e\ e & \textit{application} \\
 & | & e.\pi & \textit{operation} \\
 & & & \\
v & ::= & & \textit{values}: \\
 & | & r & \textit{resource literal} \\
 & | & \lambda x : \tau.e & \textit{abstraction}
\end{array}
\qquad
\begin{array}{lcll}
\tau & ::= & & \textit{types}: \\
 & | & \{\bar{r}\} & \\
 & | & \tau \to \tau & \\
 & & & \\
\Gamma & ::= & & \textit{type ctx}: \\
 & | & \varnothing & \\
 & | & \Gamma, x : \tau &
\end{array}
$$

Figure 3.13: Unannotated programs and types in CC.

$$
\begin{array}{lcll}
\hat{e} & ::= & & \textit{labelled exprs}: \\
 & | & x & \\
 & | & \hat{v} & \\
 & | & \hat{e}\ \hat{e} & \\
 & | & \hat{e}.\pi & \\
 & | & \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e & \textit{import} \\
 & & & \\
\hat{v} & ::= & & \textit{labelled values}: \\
 & | & r & \\
 & | & \lambda x : \hat{\tau}.\hat{e} &
\end{array}
\qquad
\begin{array}{lcll}
\hat{\tau} & ::= & & \textit{labelled types}: \\
 & | & \{\bar{r}\} & \\
 & | & \hat{\tau} \to_{\varepsilon} \hat{\tau} & \\
 & & & \\
\hat{\Gamma} & ::= & & \textit{labelled type ctx}: \\
 & | & \varnothing & \\
 & | & \hat{\Gamma}, x : \hat{\tau} &
\end{array}
$$

Figure 3.14: Annotated programs and types in CC.

The new form is $\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e$. e is some unannotated code, encapsulated by annotated code. $\hat{e}$ is the capability being given to it from the surrounding annotated environment. Inside the unannotated code, it is given the name x. ε is the maximum authority that e is allowed to exercise. As an example, suppose an unannotated `Logger`, which requires `File`, is expected to only `append` to a file, but has an implementation that writes. This would be modelled by the expression $\texttt{import}(\texttt{File.append})\ x = \texttt{File}\ \texttt{in}\ \texttt{x.write}$. For simplicity, we assume only one capability is being passed into e. Generalising so that multiple capabilities can be passed in is straightforward, but we stick with a single import to reduce formal clutter.

`import` is the only way to mix annotated and unannotated code, because it is the only situation in which we can say something interesting about the unannotated code. For the rest of our discussion on CC, we will only be interested in unannotated programs if they are encapsulated by annotated code inside an `import` expression.

3.2.2 CC Dynamic Rules

Different approaches might be taken to define the small-step semantics of CC: one is to define reductions for both annotated and unannotated programs, but this clutters the formalism with irrelevant, uninteresting rules. Another is to define reductions for either of the two and translate programs into the appropriate form before execution. Because our static rules focus on what can be said about unannotated code nested inside annotated code, we take the second approach: reductions are defined on annotated forms, and unannotated forms nested inside annotated code are transformed at runtime.

Excluding `import`, the annotated sublanguage of `CC` is the same as `OC`, so we take the reduction rules of `OC` as also being reduction rules in `CC`. For brevity, they are not restated. The new rules in `CC` are for reducing `import` expressions. The idea is that when a piece of unannotated code e is encountered inside annotated code, the surrounding `import` has its selected authority ε, so we can annotate e with ε to wrangle it into a form that can be further reduced by the rules from `OC`. To this end, we define $\texttt{annot}(e, \varepsilon)$ in Figure 3.15, which recursively annotates the parts of e with ε. There are versions of `annot` defined for expressions and types. We need to annotate contexts later, so the definition is given here. It is worth mentioning that `annot` operates on a purely syntatic level — nothing prevents us from annotating a program with something that is type unsafe, so any use of `annot` must be justified.

$\texttt{annot} :: e \times \varepsilon \to \hat{e}$

$$\begin{aligned}
&\texttt{annot}(r, _) = r \\
&\texttt{annot}(\lambda x : \tau_1.e, \varepsilon) = \lambda x : \texttt{annot}(\tau_1, \varepsilon).\texttt{annot}(e, \varepsilon) \\
&\texttt{annot}(e_1\ e_2, \varepsilon) = \texttt{annot}(e_1, \varepsilon)\ \texttt{annot}(e_2, \varepsilon) \\
&\texttt{annot}(e_1.\pi, \varepsilon) = \texttt{annot}(e_1, \varepsilon).\pi
\end{aligned}$$

$\texttt{annot} :: \tau \times \varepsilon \to \hat{\tau}$

$$\begin{aligned}
&\texttt{annot}(\{\bar{r}\}, _) = \{\bar{r}\} \\
&\texttt{annot}(\tau \to \tau, \varepsilon) = \tau \to_\varepsilon \tau.
\end{aligned}$$

$\texttt{annot} :: \Gamma \times \varepsilon \to \hat{\Gamma}$

$$\begin{aligned}
&\texttt{annot}(\varnothing, _) = \varnothing \\
&\texttt{annot}(\Gamma, x : \tau, \varepsilon) = \texttt{annot}(\Gamma, \varepsilon), x : \texttt{annot}(\tau, \varepsilon)
\end{aligned}$$

Figure 3.15: Definition of `annot`.

Before giving the dynamic rules we must update `substitution`. Because our dynamic rules are defined on annotated programs, so too will `substitution` be defined. Except for the new case for `import` expressions given in Figure 3.16, the definition is the same from `OC`. We still require that variables be only replaced with values, to prevent the introduction of arbitrary effects.

$\texttt{substitution} :: \hat{\texttt{e}} \times \hat{\texttt{v}} \times \hat{\texttt{v}} \to \hat{\texttt{e}}$

$$[\hat{v}/y](\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e) = \texttt{import}(\varepsilon)\ x = [\hat{v}/y]\hat{e}\ \texttt{in}\ e$$

Figure 3.16: New case for `substitution` in `CC`.

The new single-step reductions on `import` expressions are given in 3.17. E-IMPORT1 reduces the definition of the capability being imported. If the capability being imported is a value $\hat{v}$, then E-IMPORT2 annotates e with its selected authority ε — this is $\texttt{annot}(e, \varepsilon)$. The name of the capability x is then replaced with its definition $\hat{v}$ — this is $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$. The single-step incurs no effects. Multi-step reductions are defined the same as in `OC`, so we do not restate them.

3.2.3 `CC` Static Rules

A term might be annotated or unannotated, so we need to be able to recognise when either is well-typed. We do not reason about the effects of unannotated code directly, so judgements

$\boxed{\hat{e} \longrightarrow \hat{e} \mid \varepsilon}$

$$\frac{\hat{e} \longrightarrow \hat{e}' \mid \varepsilon'}{\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e \longrightarrow \texttt{import}(\varepsilon)\ x = \hat{e}'\ \texttt{in}\ e \mid \varepsilon'}\ \text{(E-IMPORT1)}$$

$$\frac{}{\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon) \mid \varnothing}\ \text{(E-IMPORT2)}$$

Figure 3.17: New single-step reductions in CC.

about them take the form $\Gamma \vdash e : \tau$. The subtyping judgement for unannotated code takes the form $\tau <: \tau$. A summary of these typing and subtyping rules is given in 3.18; each is analogous to some rule in OC, but the parts relating to effects have been removed.

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{(T-VAR)} \qquad \frac{}{\Gamma, r : \{r\} \vdash r : \{r\}}\ \text{(T-RESOURCE)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}\ \text{(T-ABS)}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_3}\ \text{(T-APP)} \qquad \frac{\Gamma \vdash e : \{\bar{r}\}}{\Gamma \vdash e.\pi : \texttt{Unit}}\ \text{(T-OPERCALL)}$$

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

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ \text{(S-ARROW)} \qquad \frac{\{\bar{r}_1\} \subseteq \{\bar{r}_2\}}{\{\bar{r}_1\} <: \{\bar{r}_2\}}\ \text{(S-RESOURCES)}$$

Figure 3.18: (Sub)typing judgements for the unannotated sublanguage of CC

Since the annotated subset of CC contains OC, all the OC rules apply, but now we put hats on everything to signify that a typing judgement is being made about annotated code inside an annotated context. This looks like $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon$. Except for notation the judgements are the same, so we shall not repeat them. The only new rule is ε-IMPORT, given in Figure 3.28, which gives the type and approximate effects of an `import` expression. This is the only way to reason about what effects might be incurred by some unannotated code. The rule is complicated, so we start with a simple version and spend the rest of the section building up to the final version of ε-IMPORT.

$$\frac{\begin{array}{c}\texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon \\ \hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \qquad \texttt{ho-safe}(\hat{\tau}, \varepsilon) \qquad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau\end{array}}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ \text{(ε-IMPORT)}$$

Figure 3.19: (Sub)typing judgements for the unannotated sublanguage of CC

To begin, typing $\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e$ in a context $\hat{\Gamma}$ requires us to know that the imported capability $\hat{e}$ is well-typed, so we add the premise $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1$. Since $x = \hat{e}$ is an import, it can be used throughout e. However, we do not want e to exercise ambient authority beyond that which has been explicitly selected, so whatever capabilities are used must be selected by the `import` expression; therefore, we require that e can be typechecked using only the binding $x : \hat{\tau}$. There is a problem though: e is unannotated and $\hat{\tau}$ is annotated, and there is

no rule for typechecking unannotated code in an annotated context. To get around this, we define a function `erase` in Figure 3.20 which removes the annotations from a type. We then add $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$ as a premise.

$\texttt{erase} :: \hat{\tau} \to \tau$

$$\begin{aligned}\texttt{erase}(\{\bar{r}\}) \\ \texttt{erase}(\hat{\tau}_1 \to_\varepsilon \hat{\tau}_2) &= \texttt{erase}(\hat{\tau}_1) \to \texttt{erase}(\hat{\tau}_2)\end{aligned}$$

Figure 3.20: Definition of `erase`.

The first version of ε-IMPORT is given in Figure 3.21. Since $\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon)$ by E-IMPORT2, it is sensible that its ascribed type would be $\texttt{annot}(\tau, \varepsilon)$, which is the type of the unannotated code, annotated with its selected authority ε. The effects of the `import` are $\varepsilon_1 \cup \varepsilon$ — the former comes from reducing the imported capability, which happens before the body of the `import` is annotated and executed, and the latter contains all the effects which the unannotated code is allowed to incur.

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

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ (\varepsilon\text{-IMPORT1})$$

Figure 3.21: A first rule for type-and-effect checking `import` expressions.

At the moment there is no relation between ε — the effects which e is allowed to incur — and those effects captured by the imported capability $\hat{e}$. Consider $\hat{e}' = \texttt{import}(\varnothing)\ x =$ `File in x.write`, which imports a `File` and writes to it, but declares its authority as $\varnothing$. According to ε-IMPORT1, $\vdash \hat{e}' :$ `Unit with` $\varnothing$, but this is clearly wrong since $\hat{e}'$ writes to `File`. An `import` should only be well-typed if the capability being import only captures effects that are in the unannotated code's selected authority ε. To this end we define a function `effects`, which collects the set of effects that an annotated type captures. A first definition is given in Figure 3.22. We can then add the premise $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ to require that any imported capability must not capture authority beyond that selected in ε. The updated rule is given in Figure 3.23.

$\texttt{effects} :: \hat{\tau} \to \varepsilon$

$$\begin{aligned}\texttt{effects}(\{\bar{r}\}) &= \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\} \\ \texttt{effects}(\hat{\tau}_1 \to_\varepsilon \hat{\tau}_2) &= \texttt{effects}(\hat{\tau}_1) \cup \varepsilon \cup \texttt{effects}(\hat{\tau}_2)\end{aligned}$$

Figure 3.22: A first definition of `effects`.

The counterexample from before is now rejected by ε-IMPORT2, but there are still issues: the annotations on one import can be broken by another import. To illustrate, consider Figure 3.24 where two[1] capabilities are imported. This program imports a function `go` which, when given a `Unit` $\to_\varnothing$ `Unit` function with no effects, will execute it. The other import is `File`. The unannotated code creates a `Unit` $\to$ `Unit` function which writes to `File` and passes it to `go`, which subsequently incurs `File.write`.

In the world of annotated code it is not possible to pass a file-writing function to `go`, but because the judgement $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$ discards the annotations on `go`, and since the

[1] Our formalisation only permits a single capability to be imported, but this discussion leads to a generalisation needed for the rules be safe when multiple capabilities can be imported.

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{e} \texttt{ in } e : \texttt{annot}(\tau, \varepsilon) \texttt{ with } \varepsilon \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT2})$$

Figure 3.23: A second rule for type-and-effect checking `import` expressions.

```
import({File.*})
  go = λx: Unit →∅ Unit. x unit
  f = File
in
  go (λy: Unit. f.write)
```

Figure 3.24: Permitting multiple imports will break ε-IMPORT2.

file-writing function has type `unit` → `unit`, the unannotated world accepts it. The `import` selects $\{\texttt{File}.*\}$ as its authority, which contains `File.write`, so the approximation is actually safe at the top-level, but it contains code that violates the type signature of `go`. We want to prevent this.

If `go` had the type `Unit` $\to_{\{\texttt{File.write}\}}$ `Unit` the above example would be safe, but a modified version where a file-reading function is passed to `go` would have the same issue. `go` is only safe when it expects every effect that the unannotated code might incur; if `go` had the type `Unit` $\to_{\{\texttt{File}.*\}}$ `Unit`, then the unannotated code cannot pass it a capability with an effect it isn't already expecting, so the annotation on `go` cannot be violated. Therefore we require imported capabilities to have authority to incur the effects in ε.

To achieve greater control in how we say this, the definition of `effects` is split into two separate functions called `effects` and `ho-effects`. If values of $\hat{\tau}$ possess a capability that can be used to incur the effect $r.\pi$, then $r.\pi \in \texttt{effects}(\hat{\tau})$. If values of $\hat{\tau}$ can incur an effect $r.\pi$, but need to be given the capability by someone else in order to do that, then $r.\pi \in \texttt{ho-effects}(\hat{\tau})$.

$\texttt{effects} :: \hat{\boldsymbol{\tau}} \to \boldsymbol{\varepsilon}$

$$\begin{aligned} \texttt{effects}(\{\bar{r}\}) &= \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\} \\ \texttt{effects}(\hat{\tau}_1 \to_\varepsilon \hat{\tau}_2) &= \texttt{ho-effects}(\hat{\tau}_1) \cup \varepsilon \cup \texttt{effects}(\hat{\tau}_2) \end{aligned}$$

$\texttt{ho-effects} :: \hat{\boldsymbol{\tau}} \to \boldsymbol{\varepsilon}$

$$\begin{aligned} \texttt{ho-effects}(\{\bar{r}\}) &= \varnothing \\ \texttt{ho-effects}(\hat{\tau}_1 \to_\varepsilon \hat{\tau}_2) &= \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2) \end{aligned}$$

Figure 3.25: Effect functions.

`effects` and `ho-effects` are mutually recursive, with base cases for resource types. Any effect can be directly incurred by a resource on itself, hence $\texttt{effects}(\{\bar{r}\}) = \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$. A resource cannot be used to indirectly invoke some other effect, so $\texttt{ho-effects}(\{\bar{r}\}) = \varnothing$. The mutual recursion echoes the subtyping rule for functions. Recall that functions are contravariant in their input type and covariant in their output type. The mutual recursion here is similar: both functions recurse on the input-type using the other function, and recurse on the output-type using the same function.

In light of these new definitions, we still require $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ — unannotated code must select any effect its capabilities can incur — but we add a new premise $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau})$, stipulating that imported capabilities must select every effect they could be given by unan-

notated code. The counterexample from Figure 3.24 now rejects, because $\texttt{ho-effects}(\texttt{Unit} \to_\varnothing \texttt{Unit}) \to_\varnothing \texttt{Unit}) = \varnothing$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$, but this is *still* not sufficient! Consider $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2)$. We want *every* higher-order capability involved to be expecting ε. Expanding the definition, $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$. Let $r.\pi \in \varepsilon$ and suppose $r.\pi \in \texttt{effects}(\hat{\tau}_1)$, but $r.\pi \notin \texttt{ho-effects}(\hat{\tau}_2)$. Then $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$ is still true, but $\hat{\tau}_2$ is not expecting $r.\pi$. Unannotated code could then violate the annotations on $\hat{\tau}_2$ by causing it to invoke $r.\pi$, using the same trickery from before. The cause of the issue is that $\subseteq$ does not distribute over $\cup$. We want a relation like $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$, but which also implies $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_1)$ and $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_2)$. Figure 3.26 defines this: `safe` is a distributive version of $\varepsilon \subseteq \texttt{effects}(\hat{\tau})$ and `ho-safe` is a distributive version of $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau})$.

$\boxed{\texttt{safe}(\hat{\tau}, \varepsilon)}$

$$\frac{\{r.\pi \mid r \in \bar{r}, \pi \in \Pi\} \subseteq \varepsilon}{} \text{ (SAFE-RESOURCE)}$$

$$\frac{\varepsilon \subseteq \varepsilon' \quad \texttt{ho-safe}(\hat{\tau}_1, \varepsilon) \quad \texttt{safe}(\hat{\tau}_2, \varepsilon)}{\texttt{safe}(\hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2, \varepsilon)} \text{ (SAFE-ARROW)}$$

$\boxed{\texttt{ho-safe}(\hat{\tau}, \varepsilon)}$

$$\frac{}{\texttt{ho-safe}(\{\bar{r}\}, \varepsilon)} \text{ (HOSAFE-RESOURCE)}$$

$$\frac{\texttt{safe}(\hat{\tau}_1, \varepsilon) \quad \texttt{ho-safe}(\hat{\tau}_2, \varepsilon)}{\texttt{ho-safe}(\hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2, \varepsilon)} \text{ (HOSAFE-ARROW)}$$

Figure 3.26: Safety judgements in CC.

Note again how the mutual recursion of `safe` and `ho-safe` mimics the co(ntra)variance rules for function subtyping. Some properties are also immediate: $\texttt{safe}(\hat{\tau}, \varepsilon)$ implies $\varepsilon \subseteq \texttt{effects}(\hat{\tau})$ and $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ implies $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau})$, but the converses are not true, because the safety predicates are distributive and therefore stronger.

An amended version of ε-IMPORT is given in Figure 3.27. It contains a new premise $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ which formalises the notion that every capability which could given to a value of $\hat{\tau}$ — or any of its constituent pieces — must be expecting the effects in ε that it might be given in the unannotated code.

$\boxed{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon}$

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \qquad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon \\ \texttt{ho-safe}(\hat{\tau}, \varepsilon) \qquad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau\end{array}}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \varepsilon_1} \text{ (ε-IMPORT3)}$$

Figure 3.27: A third rule for type-and-effect checking `import` expressions.

The premises so far restrict what authority can be selected by unannotated code, but what about authority passed as a function argument? Consider the example $\hat{e} = \texttt{import}(\varnothing)\ x =$

`unit in` λf : `File. f.write`. The unannotated code selects no capabilities and returns a function which, when given `File`, incurs `File.write`. This satisfies the premises in ε-IMPORT3, but its annotated type is $\{\texttt{File}\} \to_{\varnothing} \texttt{Unit}$ — not good!

Suppose the unannotated code defines a function f, which gets annotated with ε to produce $\texttt{annot}(f, \varepsilon)$. Suppose $\texttt{annot}(f, \varepsilon)$ is invoked at a later point and incurs the effect $r.\pi$. What is the source of $r.\pi$? If $r.\pi$ was selected by the `import` expression surrounding f, it is safe for $\texttt{annot}(f, \varepsilon)$ to incur this effect. Otherwise, $\texttt{annot}(f, \varepsilon)$ may have been passed an argument which can be used to incur $r.\pi$, in which case $r.\pi$ is a higher-order effect of $\texttt{annot}(f, \varepsilon)$. If the argument is a function, then by the soundness of `OC`, it must be that $r.\pi \in \varepsilon$, or it will not typecheck. If the argument is a resource r then $\texttt{annot}(f, \varepsilon)$ may exercise $r.\pi$, which our rule does not yet account for.

We want ε to contain every effect captured by resources passed into $\texttt{annot}(f, \varepsilon)$ as arguments. We can do this by inspecting its (unannotated type) for resource sets. For example, if the unannotated code has the type $\{\texttt{File}\} \to \texttt{Unit}$, then we need $\{\texttt{File}.*\}$ in ε. To do this, we add a new premise $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon$. `ho-effects` is only defined on annotated types, so we first annotate τ with $\varnothing$. We are only inspecting the resources passed into f as an argument, so the annotations on the arrow should be ignored – annotating τ with $\varnothing$ is therefore a good choice.

We can now handle the example from before: $\texttt{import}(\varnothing)\ x = \texttt{unit in}\ \lambda f : \texttt{File}.\ \texttt{f.write}$. The unannotated code types via the judgement $x : \texttt{Unit} \vdash \lambda f : \{\texttt{File}\}.\ \texttt{f.write} : \{\texttt{File}\} \to \texttt{Unit}$. Its higher-order effects are $\texttt{ho-effects}(\texttt{annot}(\{\texttt{File}\} \to \texttt{Unit}, \varnothing)) = \{\texttt{File}.*\}$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$, so the example safely rejects.

The final version of ε-IMPORT is given in Figure 3.28.

$\boxed{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon}$

$$\frac{\begin{array}{c}\texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon \\ \hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \qquad \texttt{ho-safe}(\hat{\tau}, \varepsilon) \qquad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau\end{array}}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ (\varepsilon\text{-IMPORT})$$

Figure 3.28: The final rule for typing imports.

We can now model the example from the beginning of Section 3.2., where the `Logger` implementation selects the capabilities `File` and `Socket` and exposes an unannotated function `log` with type $\texttt{Unit} \to \texttt{Unit}$. If we applied ε-IMPORT, it would annotate `log` so it has the type $\texttt{Unit} \to_{\{\texttt{File}.*, \texttt{Socket}.*\}} \texttt{Unit}$. If an annotated `Client` only expects the `File.append` effect, the `OC` rules will reject any attempt to give `Logger` to `Client` because $\{\texttt{File}.*, \texttt{Socket}.*\}$ exceeds $\{\texttt{File.append}\}$. More detailed examples are given in Chapter 4.

3.2.4 CC Soundness

Only annotated programs can be reduced and have their effects approximated, so the statement of soundness only applies to them. The theorem statement is given below.

Theorem 12 (CC Single-step Soundness)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A\ \texttt{with}\ \varepsilon_A$ *and* $\hat{e}_A$ *is not a value, then* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$*, where* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B\ \texttt{with}\ \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $\hat{e}_B, \varepsilon, \hat{\tau}_B, \varepsilon_B$*.*

From here onwards we adopt a different convention to avoid name clashes. The selected authority of an `import` is written ε_s ("epsilon select") and the imported capability is written $\hat{e}_i$ or $\hat{v}_i$ ("e import" and "v import"), and has type τ_i and approximate effects ε_i.

The rules of OC are also rules of CC, and have been proven sound in Section 3.1.4., so we do not repeat them here. We present the same theorems and lemmas, but only discuss and prove the cases which use new rules from CC. If a lemma is new, we prove all its cases. We begin with canonical forms, which is unchanged. The substitution lemma gains an extra case, but the proof is routine.

Lemma 5 (CC Canonical Forms)**.** *Unless the rule used is* ε-SUBSUME*, the following are true:*

1. *If* $\hat{\Gamma} \vdash x : \hat{\tau}$ `with` ε *then* $\varepsilon = \varnothing$.
2. *If* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}$ `with` ε *then* $\varepsilon = \varnothing$.
3. *If* $\hat{\Gamma} \vdash \hat{v} : \{\bar{r}\}$ `with` ε *then* $\hat{v} = r$ *and* $\{\bar{r}\} = \{r\}$.
4. *If* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2$ `with` ε *then* $\hat{v} = \lambda x : \tau.\hat{e}$.

Lemma 6 (CC Substitution)**.** *If* $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}$ `with` ε *and* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}'$ `with` $\varnothing$ *then* $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}$ `with` ε.

Proof. By induction on the derivation of $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}$ `with` ε.

Case: ε-IMPORT. By definition, $[\hat{v}/y](\texttt{import}(\varepsilon_s)\ y = \hat{e}_i\ \texttt{in}\ e) = \texttt{import}(\varepsilon_s)\ y = [\hat{v}/x]\hat{e}_i\ \texttt{in}\ e$. The result follows by applying the inductive assumption to $[\hat{v}/x]\hat{e}_i$. □

The progress theorem also has an extra, routine case.

Theorem 13 (CC Progress)**.** *If* $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ `with` ε *and* $\hat{e}$ *is not a value, then* $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon$*, for some* $\hat{e}', \varepsilon$.

Proof. By induction on the derivation of $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ `with` ε.

Case: ε-IMPORT. Then $\hat{e} = \texttt{import}(\varepsilon_s)\ x = \hat{e}_i\ \texttt{in}\ e$. If $\hat{e}_i$ is a non-value then $\hat{e}$ reduces by E-IMPORT1. Otherwise $\hat{e}$ reduces by E-IMPORT2. □

The preservation theorem has an extra case for when the typing rule used is ε-IMPORT. This has two subcases, depending on whether the reduction rule used was E-IMPORT1 and E-IMPORT2. The former is straightforward to prove, but the latter is tricky; we need several lemmas to do it. Firstly, since ε_s is an upper bound on what effects can be incurred by the unannotated code, it should also be an upper bound on what effects can be incurred by the capabilities passed into the unannotated code; therefore, if we take $\hat{\tau}_i$ and replace its annotations with ε_s, we should get a more general function type $\hat{\tau}_i <: \texttt{annot}(\texttt{erase}(\hat{\tau}_i), \varepsilon)$. This result is given as the pair of lemmas below.

Lemma 7 (CC Approximation 1)**.** *If* $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ *and* $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ *then* $\hat{\tau} <: \texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$.

Lemma 8 (CC Approximation 2)**.** *If* $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ *and* $\texttt{safe}(\hat{\tau}, \varepsilon)$ *then* $\texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon) <: \hat{\tau}$.

Proof. By simultaneous induction on derivations of $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ and $\texttt{safe}(\hat{\tau}, \varepsilon)$. □

Recall that function types are contravariant in their input, so the subtyping and subsetting relations flip direction when considering the input type of a function. This is why there are two lemmas: one for each direction.

Now, if E-IMPORT2 is applied, the reduction has the form $\texttt{import}(\varepsilon_s)\ x = \hat{v}_i\ \texttt{in}\ e \longrightarrow [\hat{v}_i/x]\texttt{annot}(e, \varepsilon_s) \mid \varnothing$. Since $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$, it is reasonable to expect that (1) $\hat{\Gamma} \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)$ `with` ε_s would be true, because although $\texttt{annot}(e, \varepsilon_s)$ has annotations and e does not, annotations do not change runtime semantics — the two programs have the same structure and capture the same effects. If judgement (1) holds, then $\hat{\Gamma} \vdash [\hat{v}_i/x]\texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)$ `with` ε_s would hold by the substitution lemma. That judgement (1) does hold is the subject of the following lemma.

Lemma 9 (CC Annotation)**.** *If the following are true:*

1. $\hat{\Gamma} \vdash \hat{v}_i : \hat{\tau}_i$ `with` $\varnothing$
2. $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e : \tau$
3. $\texttt{effects}(\hat{\tau}_i) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \cup \texttt{effects}(\texttt{annot}(\Gamma, \varnothing)) \subseteq \varepsilon_s$
4. $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s)$

Then $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)$ `with` ε_s.

The premises of the lemma are very specific to the premises of ε-IMPORT, but generalised to accommodate a proof by induction: e is allowed to typecheck with bindings in Γ, so long as Γ does not introduce any resources whose authority is not already in ε_s. We need Γ to keep track of effects introduced by function arguments. For example, typechecking `f.write` requires a binding for f, but $\lambda f : \{\texttt{File}\}$. `f.write` does not. Proving the lemma requires us to inductively step into the bodies of functions, at which point we need to keep track of what has been bound at that point — to do this, we permit e to typecheck in a larger environment Γ. We stipulate $\texttt{effects}(\texttt{annot}(\Gamma, \varnothing)) \subseteq \varepsilon_s$ so that any effects captured by Γ are not ambient. Note that when $\Gamma = \varnothing$ we have exactly the premises of ε-IMPORT. When we apply the annotation lemma in the proof of preservation, we choose $\Gamma = \varnothing$.

The proof of the annotation lemma is quite long, but a sketch for each case is given below.

Proof. By induction on derivations of $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e : \tau$.

Case: T-VAR. Then $e = x$. If $x \neq y$ use ε-VAR and ε-SUBSUME. Otherwise $x = y$. Then $y : \texttt{erase}(\hat{\tau}_i) \vdash x : \tau$ implies that $\hat{\tau}_i = \tau$. Apply the approximation lemma and simplify to obtain $\hat{\tau}_i <: \texttt{annot}(\tau_i, \varepsilon_s)$, and then use ε-SUBSUME to get the result.

Case: T-RESOURCE. Use ε-RESOURCE and ε-SUBSUME.

Case: T-ABS. Use inversion to get a judgement for the body of the function $\Gamma, y : \texttt{erase}(\hat{\tau}_i), x : \tau_2 \vdash e_{body} : \tau_3$ `with` ε_s. Apply the inductive hypothesis to e_{body} with $\Gamma, x : \tau_2$ as the context in which e_{body} typechecks, noting the premises for the inductive application are satisfied because $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s$ implies $\texttt{effects}(\texttt{annot}(\tau_2, \varnothing) \subseteq \varepsilon_s$. Then use ε-ABS and ε-SUBSUME.

CASE: T-APP. Apply the inductive assumption to the subexpressions, then use ε-APP and simplify.

CASE: T-OPERCALL. Apply the inductive hypothesis to the receiver and use ε-OPERCALL. This gives the approximate effects $\varepsilon_s \cup \{\bar{r}.\pi\}$. Consider where the binding for $\{\bar{r}\}$ is in $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}$ and conclude that $\{\bar{r}.\pi\} \subseteq \varepsilon_s$.

□

Armed with the annotation lemma, we can now prove the preservation theorem.

Theorem 14 (CC Preservation)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A *and* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$*, then* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B$ `with` ε_B*, where* $\hat{e}_B <: \hat{e}_A$ *and* $\varepsilon \cup \varepsilon_B \subseteq \varepsilon_A$*, for some* $\hat{e}_B, \varepsilon, \hat{\tau}_B, \varepsilon_B$.

Proof. By induction on $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A, and then on $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$.

Case: ε-IMPORT. Then $e_A = \texttt{import}(\varepsilon)\ x = \hat{e}$ `in` e. If the reduction rule used was E-IMPORT1 then the result follows by applying the inductive hypothesis to $\hat{e}$.

Otherwise $\hat{e}$ is a value and the reduction used was E-IMPORT2. The following are true:

1. $e_A = \texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e$
2. $\hat{\Gamma} \vdash e_A : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \varepsilon_1$
3. $\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon) \mid \varnothing$
4. $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}\ \texttt{with}\ \varnothing$
5. $\varepsilon = \texttt{effects}(\hat{\tau})$
6. $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$
7. $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$

Apply the annotation lemma with $\Gamma = \varnothing$ to get $\hat{\Gamma}, x : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon$. From assumption (4) we know $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}\ \texttt{with}\ \varnothing$, so the substitution lemma may be applied, giving $\hat{\Gamma} \vdash [\hat{v}/x]\texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon$. By canonical forms, $\varepsilon_1 = \varepsilon = \varnothing$. Then $\varepsilon_B = \varepsilon = \varepsilon_A \cup \varepsilon$. By examination, $\tau_A = \tau_B = \texttt{annot}(\tau, \varepsilon)$. □

From progress and preservation we can prove the single-step and multi-step soundness theorems for CC. The proofs are identical to the ones in OC.

Theorem 15 (CC Single-step Soundness)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A\ \texttt{with}\ \varepsilon_A$ *and* $\hat{e}_A$ *is not a value, then* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$*, where* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B\ \texttt{with}\ \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $\hat{e}_B$*,* ε*,* $\hat{\tau}_B$*, and* ε_B*.*

Theorem 16 (CC Multi-step Soundness)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A\ \texttt{with}\ \varepsilon_A$ *and* $\hat{e}_A \longrightarrow^* e_B \mid \varepsilon$*, then* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B\ \texttt{with}\ \varepsilon_B$*, where* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $\hat{\tau}_B$*,* ε_B*.*

Chapter 4

Applications

In this chapter we show how CC can be used to model several practical examples. This will take the form of writing a program in a high-level, capability-safe language, translating it to an equivalent CC program, and demonstrating how the rules of CC can be applied to reason about the use of effects. The high-level examples are written in a *Wyvern*-like language. We begin by discussing how the translation from Wyvern to CC will work. This also serves as a gentle introduction to Wyvern syntax. A variety of scenarios are then explored.

4.1 Translations and Encodings

Our aim is to develop some notation to help us translate Wyvern programs into CC. Our approach will be to encode these additional rules and forms into the base language of CC; essentially, to give common patterns a short-hand, so they can be easily named and recalled. This is called *sugaring*. When these derived forms are collapsed into their underlying representation, it is called *desugaring*.

4.1.1 Unit

`Unit` is a type inhabited by exactly one value called `unit`. It conveys the absence of information; in CC. The `unit` literal is defined as $\texttt{unit} \stackrel{\text{def}}{=} \lambda x : \varnothing.\ x$. It is the same in both annotated and unannotated code. In annotated code, it has the type $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to_{\varnothing} \varnothing$, while in naked code it has the type $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to \varnothing$. While these are technically two separate types, we will not distinguish between the annotated and naked versions,referring to them both as `Unit`.

Note that `unit` is a value, and because $\varnothing$ is uninhabited (there is no empty resource literal), `unit` cannot be applied to anything. Furthermore, $\vdash \texttt{unit} : \texttt{Unit with} \varnothing$ by ε-ABS, and $\vdash \texttt{unit} : \texttt{Unit}$ by T-ABS. This leads to the derived rules in 4.1.

$\boxed{\Gamma \vdash e : \tau}$
$\boxed{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon}$

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}}\ (\text{T-UNIT}) \qquad \frac{}{\hat{\Gamma} \vdash \texttt{unit} : \texttt{Unit with} \varnothing}\ (\varepsilon\text{-UNIT})$$

Figure 4.1: Derived `Unit` rules.

Since `unit` represents the absence of information, we also use it as the type when a function either takes no argument, or returns nothing. 4.2 shows the definition of a Wyvern

function which takes no argument and returns nothing, and its corresponding representation in CC.

```
def method():Unit
  unit
```

```
λx:Unit. unit
```

Figure 4.2: Desugaring of functions which take no arguments or return nothing.

4.1.2 Let

The expression `let` $x = \hat{e}_1$ `in` $\hat{e}_2$ first binds reduces $\hat{e}_1$ to a value $\hat{v}_1$, binds it to the name x in $\hat{e}_2$, and then executes $[\hat{v}_1/x]\hat{e}_2$. If $\hat{\Gamma} \vdash \hat{e}_1 : \hat{\tau}_1$ `with` ε_1, then `let` $x = \hat{e}_1$ `in` $\hat{e}_2 \stackrel{\text{def}}{=} (\lambda x : \hat{\tau}_1.\hat{e}_2)\hat{e}_1$[1]. If $\hat{e}_1$ is a non-value, we can reduce the `let` by E-APP2. If $\hat{e}_1$ is a value, we may apply E-APP3, which binds $\hat{e}_1$ to x in $\hat{e}_2$. This is fundamentally a lambda application, so it can be typed using ε-APP (or T-APP, if the terms involved are unlabelled). The new rules in 4.3 capture these derivations.

$\boxed{\Gamma \vdash e : \tau}$
$\boxed{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon}$
$\boxed{\hat{e} \to \hat{e} \mid \varepsilon}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ (\varepsilon\text{-LET})$$

$$\frac{\hat{\Gamma} \vdash \hat{e}_1 : \hat{\tau}_1\ \texttt{with}\ \varepsilon_1 \quad \hat{\Gamma}, x : \hat{\tau}_1 \vdash \hat{e}_2 : \hat{\tau}_2\ \texttt{with}\ \varepsilon_2}{\hat{\Gamma} \vdash \texttt{let}\ x = \hat{e}_1\ \texttt{in}\ \hat{e}_2 : \hat{\tau}_2\ \texttt{with}\ \varepsilon_1 \cup \varepsilon_2}\ (\varepsilon\text{-LET})$$

$$\frac{\hat{e}_1 \longrightarrow \hat{e}_1' \mid \varepsilon_1}{\texttt{let}\ x = \hat{e}_1\ \texttt{in}\ \hat{e}_2 \longrightarrow \texttt{let}\ x = \hat{e}_1'\ \texttt{in}\ \hat{e}_2 \mid \varepsilon_1}\ (\text{E-LET1})$$

$$\frac{}{\texttt{let}\ x = \hat{v}\ \texttt{in}\ \hat{e} \longrightarrow [\hat{v}/x]\hat{e} \mid \varnothing}\ (\text{E-LET2})$$

Figure 4.3: Derived `let` rules.

4.1.3 Modules and Objects

Wyvern's modules are first-class and desugar into objects; invoking a method inside a module is no different from invoking an object's method. There are two kinds of modules: pure and resourceful. For our purposes, a pure module is one with no (transitive) authority over any resources, while a resource module has (transitive) authority over some resource. A pure module may still be given a capability, for example by requesting it in a function signature, but it may not possess or capture the capability for longer than the duration of the method call. 4.4 shows an example of two modules, one pure and one resourceful, each declared in a separate file. Note how pure modules are declared with the `module` keyword, while resource modules are declared with `resource module`.

[1] You can also define an unannotated version of `let`, but we only need the annotated version

```
module PureMod

def tick(f: {File}):Unit with {File.append}
  f.append
```

```
resource module ResourceMod
require File

def tick():Unit with {File.append}
  File.append
```

Figure 4.4: Definition of two modules, one pure and the other resourceful.

Resource modules, like objects, must be instantiated. When they are instantiated they must be given the capabilities they require. In 4.4, `ResourceMod` requests the use of a `File` capability. 4.5 demonstrates how the two modules above would be instantiated and used. To prevent infinite regress the `File` must, at some point, be introduced into the program. This happens in a special main module. When the program begins execution, the `File` capability is passed into the program from the system environment. All these initial capabilities are modelled in `CC` as resource literals. `Main` then instantiates all the other modules in the program, propagating capabilities as it sees fit.

If a module is annotated, its function signatures will have effects annoations. For example, in Figure 4.4, `PureMod.tick` has the `File.append` annotation, which means the function should typecheck as $\{\texttt{File}\} \rightarrow_{\texttt{File.append}} \texttt{Unit}$.

```
resource module Main
require File
instantiate PureMod
instantiate ResourceMod(File)

PureMod.tick(File)
```

Figure 4.5: The `Main` module which instantiates `PureMod` and `ResourceMod` and then invokes `PureMod.tick`.

Before explaining our translation of Wyvern programs into `CC`, we must explain several simplifications made in all our examples which enable our particular desugaring. The only objects we use in the Wyvern programs are modules. Modules only ever contain exactly one function and the capabilities they require; they have no mutable fields. There are no self-referencing modules or recursive function definitions. Modules will not reference each other cyclically. In practice, we want allow of these things, but the simplifications minimise the constructs needed to translate Wyvern programs into `CC`, and the examples still demonstrate interesting behaviour.

Because modules do not exercise self-reference and only contain one function definition, they will be modelled as functions in `CC`. Applying this function will be equivalent to applying the single function definition in the module. A collection of modules is desugared into `CC` as follows. First, a sequence of let-bindings are used to name constructor functions which, when given the capabilities requested by a module, will return an instance of the module. If the module does not require any capabilities it will take `Unit` as its argument. The constructor function for `M` is called `MakeM`. A function is then defined which represents the body of code in the `Main` module. When invoked, this function will instantiate all the

modules by invoking their constructor functions, and then execute the body of code in main. Finally, this function is invoked with the primitive capabilities passed into `Main`.

To demonstrate this process, 4.6 shows how the examples above desugar. Lines 1-3 define the constructor for `PureMod`; since `PureMod` requires no capabilities, the constructor takes `Unit` as an argument on line 2. Lines 6-8 define the constructor for `ResourceMod`; it requires a `File` capability, so the constructor takes {`File`} as its input type on line 7. The entry point to the program is defined on lines 11-16, which invokes the constructors and then runs the body of the `main` method. Line 17 starts everything off by invoking `Main` with the initial set of capabilities, which in this case is just `File`.

```
1  let MakePureMod =
2    λx:Unit.
3      λf:{File}. f.append
4  in
5
6  let MakeResourceMod =
7    λf:{File}.
8      λx:Unit. f.append
9  in
10
11 let MakeMain =
12   λf:{File}.
13     λx: Unit.
14       let PureMod = (MakePureMod unit) in
15       let ResourceMod = (MakeResourceMod f) in
16       (ResourceMod unit)
17
18 (MakeMain File) unit
```

Figure 4.6: Desugaring of `PureMod` and `ResourceMod` into `CC`.

When an unannotated module is translated into `CC`, the desugared contents will be encapsulated with an `import` expression. The selected authority on the `import` expression will be that we expect of the unannotated code according to the principle of least authority in the particular example under consideration. For example, if the client only expects the unannotated code to have the `File.append` effect and executes some unannotated code e, the corresponding `import` expression will select `File.append` for e.

4.2 Examples

We now present several examples to show the capability-based design of `CC` can assist in reasoning about the effects and behaviour of a program. Each example exhibits some unsafe behaviour or demonstrates a particular development story. For each example we present high-level Wyvern code, translate it into `CC` using the techniques from the previous section, and then sketch an explanation as to why `CC` will accept or reject the program.

4.2.1 API Violation

In the first example there is a single primitive capability `File`. A `Logger` module possessing this capability exposes a single function `log` which incurs `File.write` when executed. The `Client` module possesses the `Logger` module and exposes a single function `run` which in-

vokes `Logger.log`, incurring `File.write`; however, the client's annotation is ∅, so it is not expecting any effects. Code is shown below.

```
resource module Logger
require File

def log(): Unit with {File.write} =
   File.write(``message written'')
```

```
module Client

def run(l: Logger): Unit with ∅ =
   l.log()
```

```
resource module Main
require File
instantiate Logger(File)

Client.run(Logger)
```

In this example, all code is fully annotated. A desugared version is given below. Lines 1-3 define the function which instantiates the `Logger` module. Lines 5-7 define the function which instantiates the `Client` module. Note how the client code takes as input a function of type `Unit` $\to_{\varnothing}$ `Unit`. Lines 9-14 define the implicit `Main` module, which, when given a `File`, will instantiate the other modules and execute the client code. The program begins execution on line 16, where initial capabilities (here just `File`) and arguments are passed to `Main`.

```
let MakeLogger =
  (λf: File.
    λx: Unit. let _ = f.append in f.write) in

let MakeClient =
  (λx: Unit.
    λlogger: Unit →∅ Unit. logger unit) in

let MakeMain =
  (λf: File.
    λx: Unit.
      let LoggerModule = MakeLogger f in
      let ClientModule = MakeClient unit in
      ClientModule LoggerModule) in

(MakeMain File) unit
```

At line 12, when typing `LoggerModule`, an application of ε-APP gives the judgement $\texttt{f} : \{\texttt{File}\} \vdash \texttt{LoggerModule} : \texttt{Unit} \to_{\texttt{File.write}} \texttt{Unit with} \varnothing$. At line 13 the same rule gives $\texttt{f} : \{\texttt{File}\} \vdash \texttt{ClientModule} : (\texttt{Unit} \to_{\varnothing} \texttt{Unit}) \to \texttt{Unit with} \varnothing$. Now at line 14, when trying to apply ε-APP, there is a type mismatch because the formal argument of `ClientModule` expects a function with no effects, but `LoggerModule` has typed as incurring `File.write`. This example safely rejects.

4.2.2 Unannotated Client

This example is a modification of the previous one. Now the `Client` is unannotated. If the client code is executed, what effects will it have? For the developer maintaining `Client`,

the answer is not immediately clear because it depends on what effects are incurred by `Logger.log`. The code for this example is given below.

```
resource module Logger
require File

def log(): Unit with File.append =
   File.append(``message logged'')
```

```
module Client
require Logger

def run(): Unit =
  Logger.log()
```

```
resource module Main
require File
instantiate Logger(File)
instantiate Client(Logger)

Client.run()
```

The desugared version is given below. It first creates two functions, `MakeLogger` and `MakeClient`, which instantiate the `Logger` and `Client` modules; the client code is treated as an implicit module. Lines 1-4 define a function which, given a `File`, returns a record containing a single `log` function. Lines 6-8 define a function which, given a `Logger`, returns the unannotated client code, wrapped inside an `import` expression selecting its needed authority. Lines 10-14 define `MakeMain` which returns the implicit main module that, when executed, instantiates all the other modules in the program and invokes the code in `Main`. Program execution begins on line 16, where `Main` is given the initial capabilities — which, in this case, is just `File`.

```
1  let MakeLogger =
2    (λf: File.
3      λx: Unit. f.append) in
4
5  let MakeClient =
6    (λlogger: Logger.
7      import(File.append) logger = logger in
8        λx: Unit. logger unit) in
9
10 let MakeMain =
11   (λf: File.
12     λx: Unit.
13       let LoggerModule = MakeLogger f in
14       let ClientModule = MakeClient LoggerModule in
15       ClientModule unit) in
16
17 (MakeMain File) unit
```

The interesting part is on lines 7-8, where the unannotated code selects `File.append` as its authority. This is exactly the effects of the logger, i.e. $\texttt{effects}(\texttt{Unit} \to_{\texttt{File.append}} \texttt{Unit}) = \{\texttt{File.append}\}$. The code also satisfies the higher-order safety predicates, and the body of the `import` expression typechecks in the empty context. Therefore, the unannotated code typechecks by ε-IMPORT.

4.2.3 Unannotated Library

The next example inverts the roles of the last scenario: now, the annotated `Client` wants to use the unannotated `Logger`. The `Logger` module captures the `File` capability, and exposes a single function `log` with the `File.append` effect. However, the `Client` has a function `run` which executes `Logger.log`, incurring the effect of `File.append`, but declares its set of effects as $\varnothing$, so the implementation and signature of `Client.run` are inconsistent.

```
resource module Logger
require File

def log(): Unit =
   File.append(``message logged'')
```

```
resource module Client
require Logger

def run(): Unit with ∅ =
  Logger.log()
```

```
resource module Main
require File
instantiate Logger(File)
instantiate Client(Logger)

Client.run()
```

A desugaring is given below. Lines 1-3 define the function which instantiates the `Logger` module. Lines 5-8 define the function which instantiates the `Client` module. Lines 10-15 define the function which instantiates the `Main` module. Line 17 initiates the program, supplying `File` to the `Main` module and invoking its main method. On lines 3-4, the unannotated code is modelled using an `import` expression which selects $\varnothing$ as its authority. So far this coheres to the expectations of `Client`. However, ε-IMPORT cannot be applied because the name being bound, `f`, has the type $\{\texttt{File}\}$, and $\texttt{effects}(\{\texttt{File}\}) = \{\texttt{File}.*\}$, which is inconsistent with the declared effects $\varnothing$.

```
let MakeLogger =
  (λf: File.
    import(∅) f = f in
      λx: Unit. f.append) in

let MakeClient =
  (λlogger: Logger.
    λx: Unit. logger unit) in

let MakeMain =
  (λf: File.
    let LoggerModule = MakeLogger f in
    let ClientModule = MakeClient LoggerModule in
    ClientModule unit) in

(MakeMain File) unit
```

The only way for this to typecheck would be to annotate `Client.run` as having every effect on `File`. This demonstrates how the effect-system of CC approximates unannotated code: it simply considers it as having every effect which could be incurred on those resources in scope, which here is `File`.$*$.

4.2.4 Unannotated Library 2

In yet another variation of the previous examples, the `Logger` module is now passed `File` as an argument, rather than possessing it. `Logger.log` still incurs `File.append` inside the unannotated code, which causes the implementation of `Client.run` to violate its signature. Because `Logger` has no dependencies, it is now directly instantiated by `Client`.

```
module Logger

def log(f: {File}): Unit
   f.append(``message logged'')
```

```
module Client
instantiate Logger(File)

def run(f: {File}): Unit with ∅
  Logger.log(File)
```

```
resource module Main
require File
instantiate Client

Client.run(File)
```

The desugaring, given below, is slightly different in form from the previous examples because `Logger` is instantiated by `Client`. The `MakeLogger` function is defined on lines 2-6, and invoked on line 7. `MakeClient` then returns a function which, when given a `File`, invokes the function in the `Logger` module (this is `Client.run`). `Main` now only instantiates `ClientModule` on line 13 and then invokes its function on line 14, passing `File` as an argument.

The `Logger` module is a function $\lambda f : \{\texttt{File}\}.\texttt{f.append}$, but encapsulated within an `import` expression selecting its authority as $\varnothing$ on line 5, to be consistent with the expectations of `Client`. No capabilities are being passed into the `Logger`, which is represented by the import $\texttt{y} = \texttt{unit}$. However, ε-IMPORT will not accept the unannotated code in `Logger`, because it violates the premise $\varepsilon = \texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon))$. In this case, $\varepsilon = \varnothing$, but $\tau = \{\texttt{File}\} \to \texttt{Unit}$ and $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) = \{\texttt{File}.*\}$. The example safely rejects.

```
let MakeClient =
  (λx: Unit.
    let MakeLogger =
      (λx: Unit.
        import({File.*}) y=unit in
          λf: {File}. f.append) in
    let LoggerModule = MakeLogger unit in
    λf: {File}. LoggerModule f) in

let MakeMain =
  (λf: {File}.
    λx: Unit.
      let ClientModule = MakeClient unit in
      ClientModule f) in

(MakeMain File) unit
```

To make this example typecheck would require us to change the annotation on `Client.run` to be $\{\texttt{File}.*\}$; then ε-IMPORT would type the code as $\{\texttt{File}\} \to_{\texttt{File}.*} \texttt{Unit with } \{\texttt{File}.*\}$.

Note how the unannotated code, and the function it returns, are both said to have the effects $\{\texttt{File}.*\}$. It is true that the function has these effects when it is invoked, but the function literal itself being returned by the unannotated code incurs no effects, so this is a vast approximation. In fact, since the unannotated code is not directly exercising any authority, it is not able to directly incur any effect. If the function it returns is never used, then the program might not incur any effects on `File`, but the approximation here always treats it as though every effect on `File` is incurred.

4.2.5 Higher-Order Effects

In this scenario, `Main` instantiates the `Plugin` module, which itself instantiates the `Malicious` module. `Plugin` exposes a single function `run`, should incur no effects. However, the implementation tries to read from `File` by wrapping the operation inside a function and passing it to `Malicious`, where `File.read` incurs in a higher-order manner.

```
module Malicious

def log(f: Unit → Unit):Unit
  f()
```

```
module Plugin
instantiate Malicious

def run(f: {File}): Unit with ∅
  Malicious.log(λx:Unit. f.read)
```

```
resource module Main
require File
instantiate Plugin

Plugin.run(File)
```

This examples shows how higher-order effects can obfuscate potential security risks. On line 3 of `Malicious`, the argument to `log` has type `Unit → Unit`. This will only typecheck using the T-rules from the unannotated fragment of CC; no approximation is made inside `Malicious`. The type `Unit → Unit` says nothing about the effects which might incur from executing this function. It is not clear from inspecting the unannotated code that it is doing something malicious. To realise this requires one to examine both `Plugin` and `Malicious`.

A desugared version is given below. On lines 5-6, the `Malicious` code selects its authority as $\varnothing$, to be consistent with the annotation on `Plugin.run`. For the same reasons as in the previous section, this example is rejected by ε-IMPORT, because the higher-order effects of $\texttt{ˇf} : \{\texttt{File}\}$. `LoggerModule f` are $\texttt{File}.*$, which is not contained in the selected authority.

```
let MakePlugin =
  (λx: Unit.
    let MakeMalicious =
      (λx: Unit.
        import(∅) y=unit in
          λf: {File}. f.append) in
    let LoggerModule = MakeLogger unit in
    λf: {File}. LoggerModule f) in

let MakeMain =
  (λf: {File}.
    λx: Unit.
      let ClientModule = MakeClient unit in
```

```
14        ClientModule f) in
15
16 (MakeMain File) unit
```

To get this example to typecheck, the `import` expression would have to select `File`.$*$ as its authority. The unannotated code would then typecheck as $\{\texttt{File}\} \to_{\texttt{File}.*} \texttt{Unit}$, and any application of it would be said to incur `File`.$*$ by ε-APP.

4.2.6 Resource Leak

This is another example of trying to obfuscate an unsafe effect by invoking it in a higher-order manner. The setup is the same, except the function which `Plugin` passes to `Malicious` now returns `File` when invoked. `Malicious` then incurs `File.read` by invoking its argument to get `File`, and then directly calling `read` on it.

```
1 module Malicious
2
3 def log(f: Unit → File):Unit
4   f().read
```

```
1 module Plugin
2 instantiate Malicious
3
4 def run(f: {File}): Unit with ∅
5   Malicious.log(λx:Unit. f)
```

```
1 resource module Main
2 require File
3 instantiate Plugin
4
5 Plugin.run(File)
```

The desugaring is given below. The unannotated code in `Malicious` is given on lines 5-6. The selected authority is $\varnothing$, to be consistent with the annotation on `Plugin`. Nothing is being imported, so the `import` binds a name `y` to `unit`. This example is rejected by ε-IMPORT because the premise $\varepsilon = \texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon))$ is not satisfied. In this case, $\varepsilon = \varnothing$ and $\tau = (\texttt{Unit} \to \{\texttt{File}\}) \to \texttt{Unit}$. Then $\texttt{annot}(\tau, \varepsilon) = (\texttt{Unit} \to_{\varnothing} \{\texttt{File}\}) \to_{\varnothing} \texttt{Unit}$ and $\texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon)) = \{\texttt{File}.*\}$. Thus, the premise cannot be satisfied and the example safely rejects.

```
1  let MakePlugin =
2    (λx: Unit.
3      let MakeMalicious =
4        (λx: Unit.
5          import(∅) y=unit in
6            λf: Unit → {File}. f().read) in
7      let LoggerModule = MakeLogger unit in
8      λf: {File}. LoggerModule f) in
9
10 let MakeMain =
11   (λf: {File}.
12     λx: Unit.
13       let ClientModule = MakeClient unit in
14       ClientModule f) in
15
16 (MakeMain File) unit
```

Chapter 5

Conclusions

In this report we have looked at OC, which extends $\lambda^{\rightarrow}$ with a notion of primitive capabilities and their effects. OC programs are fully annotated with their effects; relaxing this requirement, we obtained CC, which allows unannotated code to be nested inside annotated code with a new `construct`. The capability-safe design of CC allows us to make a safe inference about the effects of the unannotated code by inspecting what capabilities are passed to it by its annotated surroundings. The result is a sound, lightweight effect-system. Such an approach allows code to be incrementally annotated, giving developers a balance between safety and convenience and alleviating the verbosity that has discouraged the adoption of effect systems [21].

5.1 Related Work

Capabilities were introduced by Dennis and Van Horn as a way to control which processes in an operating system had permission to access certain parts of memory [7]. An *access control list* would declare what permissions a program may exercise. These early ideas are considerably different to the object capability model introduced by Mark Miller [16], which imposes constraint on how permissions can proliferate. Maffeis et. al. formalised the notion of a capability-safe language and showed that a subset of Caja (a Javascript implementation) is capability-safe [15]. Miller's model has also has been applied to more heavyweight formal systems: Drossopoulou et. al. combined Hoare logic with capabilities to determine whether components of a system can be trusted [9]. Other capability-safe languages include Wyvern [18] and Newspeak [3].

The original effect system by Lucassen and Gifford was used to determine if two expressions could safely run in parallel [13]. Subsequent applications include determining what functions a program might invoke [26] and what regions in memory might be accessed or updated during execution [25]. In these systems, "effects" are performed upon "regions"; in ours, "operations" are performed upon "resources". An important difference in CC is the distinction between annotated and unannotated code: only the former will type-and-effect-check. This approach allows for an effect discipline to be incrementally imposed on an otherwise effect-unconscious system.

Fengyun Liu has also combined capability-safety and effect systems, with applications to purity analysis in Scala [12]. If a function is known to be pure then optimisations such as inlining and parallelisation can be made. Liu's work is motivated by achieving such optimisations for Scala compilers. It distinguishes between free and stoic functions: free functions may exercise ambient authority whereas stoic functions may not. Stoic functions are therefore capability-safe pockets whose purity can be determined by examining what

capabilities are passed into them. Liu's System F-Impure does not track effects, whereas CC, by distinguishing between regular effects and higher-order effects, gives more fine-grained detail about what a piece of code will do when executed.

The systems by Lucassen and Liu have effect polymorphism, whereas CC does not. Another capability-safe effect-system is the one by Devriese et. al., who use effect polymorphism and possible world semantics to guarantee behavioural invariants on data structures [8]. Our approach is not as expressive, based only on a topological analysis of how capabilities can be passed around the program, but the formalism is much more lightweight.

5.2 Conclusions and Future Work

In this report we have explored, through CC, how capability-safety enables a lightweight, incremental effect system that can help developers reason about the safety and trustworthiness of components in a program.

The example from subsection 4.2.4. illustrated one case where the inferred effects of unannotated code was wildly overapproximated: $\hat{e} = \texttt{import}(\texttt{File}.*)\ x = \texttt{unit in}\ \lambda f : \{\texttt{File}\}.\ \texttt{f.write}$. The unannotated code returns a function which, when given a File, will write to it. The approximation for this code is File.$*$, but the function literal incurs no effects unless it is actually invoked. Perhaps by a refinement of the rules for higher-order effects, the approximation of CC can do better.

The conception of effects is quite narrow, only considering the invocation of operations on a primitive capability as an effect. This definition could be generalised to allow for other sorts of effects, such as accessing or writing to an object's mutable state. Resources and operations are fixed throughout runtime; it would be interesting to consider the theory in a setting where they can be created or destroyed.

The current theory contains no effect polymorphism. This would allow the type of a function to be parameterised by a set of effects. For an example of such a function, consider map: given a function f and a list l, map applies f to every element of l to produce a new list l'. The effects of map are dependent on the effects of f. The only way to define map in CC would be to conservatively approximate it as having every effect, in which case all precision has been lost. A polymorphic effect system which considers the type of map as being parameterised by a set of effects could give more meaningful approximations.

Lastly, the ideas in this paper might be extended and developed to the point where they can be used in real-world situations. Implementing these ideas in an existing, general-purpose language would do much towards that end.

Appendix A

OC Proofs

Lemma 10 (OC Canonical Forms)**.** *Unless the rule used is* ε-SUBSUME*, the following are true:*

1. *If* $\Gamma \vdash x : \tau$ `with` ε *then* $\varepsilon = \varnothing$.
2. *If* $\Gamma \vdash v : \tau$ `with` ε *then* $\varepsilon = \varnothing$.
3. *If* $\Gamma \vdash v : \{\bar{r}\}$ `with` ε *then* $v = r$ *and* $\{\bar{r}\} = \{r\}$.
4. *If* $\Gamma \vdash v : \tau_1 \to_{\varepsilon'} \tau_2$ `with` ε *then* $v = \lambda x : \tau.e$.

Proof.

1. The only rule that applies to variables is ε-VAR which ascribes the type $\varnothing$.
2. By definition a value is either a resource literal or a lambda. The only rules which can type values are ε-RESOURCE and ε-ABS. In the conclusions of both, $\varepsilon = \varnothing$.
3. The only rule ascribing the type $\{\bar{r}\}$ is ε-RESOURCE. Its premises imply the result.
4. The only rule ascribing the type $\tau_1 \to_{\varepsilon'} \tau_2$ is ε-ABS. Its premises imply the result.

□

Theorem 17 (OC Progress)**.** *If* $\Gamma \vdash e : \tau$ `with` ε *and* e *is not a value or variable, then* $e \longrightarrow e' \mid \varepsilon$*, for some* e', ε.

Proof. By induction on $\Gamma \vdash e : \tau$ `with` ε.

Case: ε-VAR, ε-RESOURCE, or ε-ABS. Then e is a value or variable and the theorem statement holds vacuously.

Case: ε-APP. Then $e = e_1\ e_2$. If e_1 is not a value or variable it can be reduced $e_1 \longrightarrow e_1' \mid \varepsilon$ by inductive assumption, so $e_1\ e_2 \longrightarrow e_1'\ e_2 \mid \varepsilon$ by E-APP1. If $e_1 = v_1$ is a value and e_2 a non-value, then e_2 can be reduced $e_2 \longrightarrow e_2' \mid \varepsilon$ by inductive assumption, so $e_1\ e_2 \longrightarrow v_1\ e_2' \mid \varepsilon$ by E-APP2. Otherwise $e_1 = v_1$ and $e_2 = v_2$ are both values. By inversion on ε-APP and canonical forms, $\Gamma \vdash v_1 : \tau_2 \to_{\varepsilon'} \tau_3$ `with` $\varnothing$, and $v_1 = \lambda x : \tau_2.e_{body}$. Then $(\lambda x : \tau.e_{body})v_2 \longrightarrow [v_2/x]e_{body} \mid \varnothing$ by E-APP3.

Case: ε-OPERCALL. Then $e = e_1.\pi$. If e_1 is a non-value it can be reduced $e_1 \longrightarrow e_1' \mid \varepsilon$ by inductive assumption, so $e_1.\pi \longrightarrow e_1'.\pi \mid \varepsilon$ by E-OPERCALL1. Otherwise $e_1 = v_1$ is a value. By inversion on ε-OPERCALL and canonical forms, $\Gamma \vdash v_1 : \{r\}$ `with` $\{r.\pi\}$, and $v_1 = r$. Then $r.\pi \longrightarrow$ `unit` $\mid \{r.\pi\}$ by E-OPERCALL2.

Case: ε-SUBSUME. If e is a value or variable, the theorem holds vacuously. Otherwise by inversion on ε-SUBSUME, $\Gamma \vdash e : \tau'$ `with` ε', and $e \longrightarrow e' \mid \varepsilon$ by inductive assumption.

□

Lemma 11 (OC Substitution). *If* $\Gamma, x : \tau' \vdash e : \tau$ `with` ε *and* $\Gamma \vdash v : \tau'$ `with` $\varnothing$ *then* $\Gamma \vdash [v/x]e : \tau$ `with` ε.

Proof. By induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$ `with` ε.

Case: ε-VAR. Then $e = y$ is a variable. Either $y = x$ or $y \neq x$. Suppose $y = x$. By applying canonical Forms to the theorem assumption $\Gamma, x : \tau' \vdash e : \tau'$ `with` $\varnothing$, hence $\tau' = \tau$. $[v/x]y = [v/x]x = v$, and by assumption, $\Gamma \vdash v : \tau'$ `with` $\varnothing$, so $\Gamma \vdash [v/x]y : \tau$ `with` $\varnothing$.

Otherwise $y \neq x$. By applying canonical forms to the theorem assumption $\Gamma, x : \tau' \vdash y : \tau$ `with` $\varnothing$, so $y : \tau \in \Gamma$. Since $[v/x]y = y$, then $\Gamma \vdash y : \tau$ `with` $\varnothing$ by ε-VAR.

Case: ε-RESOURCE. Because $e = r$ is a resource literal then $\Gamma \vdash r : \{r\}$ `with` $\varnothing$ by canonical forms. By definition $[v/x]r = r$, so $\Gamma \vdash [v/x]r : \{\bar{r}\}$ `with` $\varnothing$.

Case: ε-APP. By inversion $\Gamma, x : \tau' \vdash e_1 : \tau_2 \to_{\varepsilon_3} \tau_3$ `with` ε_A and $\Gamma, x : \tau' \vdash e_2 : \tau_2$ `with` ε_B, where $\varepsilon = \varepsilon_A \cup \varepsilon_B \cup \varepsilon_3$ and $\tau = \tau_3$. From inversion on ε-APP and inductive assumption, $\Gamma \vdash [v/x]e_1 : \tau_2 \to_{\varepsilon_3} \tau_3$ `with` ε_A and $\Gamma \vdash [v/x]e_2 : \tau_2$ `with` ε_B. By ε-APP $\Gamma \vdash ([v/x]e_1)([v/x]e_2) : \tau_3$ `with` $\varepsilon_A \cup \varepsilon_B \cup \varepsilon_3$. By simplifying and applying the definition of `substitution`, this is the same as $\Gamma \vdash [v/x](e_1\ e_2) : \tau$ `with` ε.

Case: ε-OPERCALL. By inversion $\Gamma, x : \tau' \vdash e_1 : \{\bar{r}\}$ `with` ε_1 and $\tau =$ `Unit` and $\varepsilon = \varepsilon_1 \cup \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$. By inductive assumption, $\Gamma \vdash [v/x]e_1 : \{\bar{r}\}$ `with` ε_1. Then by ε-OPERCALL, $\Gamma \vdash ([v/x]e_1).\pi :$ `Unit with` $\varepsilon_1 \cup \{r.\pi \mid r.\pi \in \bar{r} \times \Pi\}$. By simplifying and applying the definition of `substitution`, this is the same as $\Gamma \vdash [v/x](e_1.\pi) : \tau$ `with` ε.

Case: ε-SUBSUME. By inversion, $\Gamma, x : \tau' \vdash e : \tau_2$ `with` ε_2, where $\tau_2 <: \tau$ and $\varepsilon_2 \subseteq \varepsilon$. By inductive hypothesis, $\Gamma \vdash [v/x]e : \tau_2$ `with` ε_2. Then $\Gamma \vdash [v/x]e : \tau$ `with` ε by ε-SUBSUME. □

Theorem 18 (OC Preservation). *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* $e_A \longrightarrow e_B \mid \varepsilon$, *then* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, *for some* $e_B, \varepsilon, \tau_B, \varepsilon_B$.

Proof. By induction on the derivation of $\Gamma \vdash e_A : \tau_A$ `with` ε_A and then the derivation of $e_A \longrightarrow e_B \mid \varepsilon$.

Case: ε-VAR, ε-RESOURCE, ε-UNIT, ε-ABS. Then e_A is a value and cannot be reduced, so the theorem holds vacuously.

Case: ε-APP. Then $e_A = e_1\ e_2$ and $\Gamma \vdash e_1 : \tau_2 \longrightarrow_{\varepsilon_3} \tau_3$ `with` ε_1 and $\Gamma \vdash e_2 : \tau_2$ `with` ε_2 and $\tau_B = \tau_3$ and $\varepsilon_A = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$. In each case we choose $\tau_B = \tau_A$ and $\varepsilon_B \cup \varepsilon = \varepsilon_A$.

Subcase: E-APP1. Then $e_1\ e_2 \longrightarrow e_1'\ e_2 \mid \varepsilon$. By inversion on E-APP1, $e_1 \longrightarrow e_1' \mid \varepsilon$. By inductive hypothesis and ε-SUBSUME $\Gamma \vdash v_1 : \tau_2 \longrightarrow_{\varepsilon_3} \tau_3$ `with` ε_1. Then $\Gamma \vdash e_1'\ e_2 : \tau_3$ `with` $\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$ by ε-APP.

Subcase: E-APP2. Then $e_1 = v_1$ is a value and $e_2 \longrightarrow e_2' \mid \varepsilon$. By inversion on E-APP2, $e_2 \longrightarrow e_2' \mid \varepsilon$. By inductive hypothesis and ε-SUBSUME $\Gamma \vdash e_2' : \tau_2$ `with` ε_2. Then $\Gamma \vdash v_1\ e_2' : \tau_3$ `with` $\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$ by ε-APP.

Subcase: E-APP3. Then $e_1 = \lambda x : \tau_2.e_{body}$ and $e_2 = v_2$ are values and $(\lambda x : \tau_2.e_{body})\ v_2 \longrightarrow [v_2/x]e_{body} \mid \varnothing$. By inversion on the rule ε-APP used to type $\lambda x : \tau_2.e_{body}$, we know $\Gamma, x : \tau_2 \vdash$

$e_{body} : \tau_3$ `with` ε_3. $e_1 = v_1$ and $e_2 = v_2$ are values, so $\varepsilon_1 = \varepsilon_2 = \varnothing$ by canonical forms . Then by the substitution lemma, $\Gamma \vdash [v_2/x]e_{body} : \tau_3$ `with` ε_3 and $\varepsilon_A = \varepsilon_B = \varepsilon$.

Case: ε-OPERCALL. Then $e_A = e_1.\pi$ and $\Gamma \vdash e_1 : \{\bar{r}\}$ `with` ε_1 and $\tau_A =$ `Unit` and $\varepsilon_A = \varepsilon_1 \cup \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$.

Subcase: E-OPERCALL1. Then $e_1.\pi \longrightarrow e_1'.\pi \mid \varepsilon$. By inversion on E-OPERCALL1, $e_1 \longrightarrow e_1' \mid \varepsilon$. By inductive hypothesis and application of ε-SUBSUME, $\Gamma \vdash e_1' : \{\bar{r}\}$ `with` ε_1. Then $\Gamma \vdash e_1'.\pi : \{\bar{r}\}$ `with` $\varepsilon_1 \cup \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$ by ε-OPERCALL.

Subcase: E-OPERCALL2. Then $e_1 = r$ is a resource literal and $r.\pi \longrightarrow$ `unit` $\mid \{r.\pi\}$. By canonical forms, $\varepsilon_1 = \varnothing$. By ε-UNIT, $\Gamma \vdash$ `unit` : `Unit` `with` $\varnothing$. Therefore $\tau_B = \tau_A$ and $\varepsilon \cup \varepsilon_B = \{r.\pi\} = \varepsilon_A$. □

Theorem 19 (`OC` Single-step Soundness)**.** *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* e_A *is not a value, then* $e_A \longrightarrow e_B \mid \varepsilon$*, where* $\Gamma \vdash e_B : \tau_B$ `with` ε_B *and* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $e_B, \varepsilon, \tau_B, \varepsilon_B$*.*

Proof. If e_A is not a value then the reduction exists by the progress theorem. The rest follows by the preservation theorem. □

Theorem 20 (`OC` Multi-step Soundness)**.** *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* $e_A \longrightarrow^* e_B \mid \varepsilon$*, where* $\Gamma \vdash e_B : \tau_B$ `with` ε_B *and* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*.*

Proof. By induction on the length of the multi-step reduction.

Case: Length 0. Then $e_A = e_B$ and $\tau_A = \tau_B$ and $\varepsilon = \varnothing$ and $\varepsilon_A = \varepsilon_B$.

Case: Length $n + 1$. By inversion the multi-step can be split into a multi-step of length n, which is $e_A \longrightarrow^* e_C \mid \varepsilon'$, and a single-step of length 1, which is $e_C \longrightarrow e_B \mid \varepsilon''$, where $\varepsilon = \varepsilon' \cup \varepsilon''$. By inductive assumption and preservation theorem, $\Gamma \vdash e_C : \tau_C$ `with` ε_C and $\Gamma \vdash e_B : \tau_B$ `with` ε_B, where $\tau_C <: \tau_A$ and $\varepsilon_C \cup \varepsilon' \subseteq \varepsilon_A$. By single-step soundness, $\tau_B <: \tau_C$ and $\varepsilon_B \cup \varepsilon'' \subseteq \varepsilon_C$. Then by transitivity, $\tau_B <: \tau$ and $\varepsilon_B \cup \varepsilon' \cup \varepsilon'' = \varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$. □

Appendix B

CC Proofs

Lemma 12 (CC Canonical Forms)**.** *Unless the rule used is* ε-SUBSUME*, the following are true:*

1. *If* $\hat{\Gamma} \vdash x : \hat{\tau}$ `with` ε *then* $\varepsilon = \varnothing$.
2. *If* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}$ `with` ε *then* $\varepsilon = \varnothing$.
3. *If* $\hat{\Gamma} \vdash \hat{v} : \{\bar{r}\}$ `with` ε *then* $\hat{v} = r$ *and* $\{\bar{r}\} = \{r\}$.
4. *If* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2$ `with` ε *then* $\hat{v} = \lambda x : \tau.\hat{e}$.

Proof. Same as for OC. □

Theorem 21 (CC Progress)**.** *If* $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ `with` ε *and* $\hat{e}$ *is not a value, then* $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon$*, for some* $\hat{e}', \varepsilon$.

Proof. By induction on the derivation of $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ `with` ε.

Case: ε-MODULE. Then $\hat{e} = \texttt{import}(\varepsilon_s)\ x = \hat{e}_i\ \texttt{in}\ e$. If $\hat{e}_i$ is a non-value then $\hat{e}_i \longrightarrow \hat{e}_i' \mid \varepsilon$ by inductive assumption and $\texttt{import}(\varepsilon_s)\ x = \hat{e}_i\ \texttt{in}\ e \longrightarrow \texttt{import}(\varepsilon_s)\ x = \hat{e}_i'\ \texttt{in}\ e \mid \varepsilon$ by E-MODULE1. Otherwise $\hat{e}_i = \hat{v}_i$ is a value and $\texttt{import}(\varepsilon_s)\ x = \hat{v}_i\ \texttt{in}\ e \longrightarrow [\hat{v}_i/x]\texttt{annot}(e, \varepsilon_s) \mid \varnothing$ by E-MODULE2. □

Lemma 13 (CC Substitution)**.** *If* $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}$ `with` ε *and* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}'$ `with` $\varnothing$ *then* $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_A : \hat{\tau}$ `with` ε.

Proof. By induction on the derivation of $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}$ `with` ε.

Case: ε-MODULE. Then the following are true.

1. $\hat{e} = \texttt{import}(\varepsilon_s)\ x = \hat{e}_i\ \texttt{in}\ e$
2. $\hat{\Gamma}, y : \hat{\tau}' \vdash \hat{e}_i : \hat{\tau}_i$ `with` ε_i
3. $y : \texttt{erase}(\hat{\tau}_i) \vdash e : \tau$
4. $\hat{\Gamma}, y : \hat{\tau}' \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}_i\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s)$ `with` $\varepsilon_s \cup \varepsilon_i$
5. $\varepsilon_s = \texttt{effects}(\hat{\tau}_i) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing))$
6. $\hat{\tau}_A = \texttt{annot}(\tau, \varepsilon)$
7. $\hat{\varepsilon}_A = \varepsilon_s \cup \varepsilon_i$

By applying inductive assumption to (2) $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_i : \hat{\tau}_i$ `with` ε_i. Then by ε-MODULE $\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ y = [\hat{v}/x]\hat{e}_i\ \texttt{in}\ e : \texttt{annot}(\tau_i, \varepsilon_s)$ `with` $\varepsilon_s \cup \varepsilon_i$. By definition of `substitution`, the form in this judgement is the same as $[\hat{v}/x]\hat{e}$. □

Lemma 14 (CC Approximation 1)**.** *If* $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ *and* $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ *then* $\hat{\tau} <: \texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$.

Lemma 15 (CC Approximation 2)**.** *If* $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ *and* $\texttt{safe}(\hat{\tau}, \varepsilon)$ *then* $\texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon) <: \hat{\tau}$.

Proof. By simultaneous induction on derivations of `safe` and `ho-safe`.

Case: $\hat{\tau} = \{\bar{r}\}$ Then $\hat{\tau} = \texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$ and the results for both lemmas hold immediately.

Case: $\hat{\tau} = \hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2$, $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ It is sufficient to show $\hat{\tau}_2 <: \texttt{annot}(\texttt{erase}(\hat{\tau}_2), \varepsilon)$ and $\texttt{annot}(\texttt{erase}(\hat{\tau}_1), \varepsilon) <: \hat{\tau}_1$, because the result will hold by S-EFFECTS. To achieve this we shall inductively apply **lemma 1** to $\hat{\tau}_2$ and **lemma 2** to $\hat{\tau}_1$.

From $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ we have $\texttt{ho-effects}(\hat{\tau}_1) \cup \varepsilon' \cup \texttt{effects}(\hat{\tau}_2) \subseteq \varepsilon$ and therefore $\texttt{effects}(\hat{\tau}_2) \subseteq \varepsilon$. From $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ we have $\texttt{ho-safe}(\hat{\tau}_2, \varepsilon)$. Therefore we can apply **lemma 1** to $\hat{\tau}_2$.

From $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ we have $\texttt{ho-effects}(\hat{\tau}_1) \cup \varepsilon' \cup \texttt{effects}(\hat{\tau}_2) \subseteq \varepsilon$ and therefore $\texttt{ho-effects}(\hat{\tau}_1) \subseteq \varepsilon$. From $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ we have $\texttt{ho-safe}(\hat{\tau}_1, \varepsilon)$. Therefore we can apply **lemma 2** to $\hat{\tau}_1$.

Case: $\hat{\tau} = \hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2$, $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{safe}(\hat{\tau}, \varepsilon)$ It is sufficient to show $\texttt{annot}(\texttt{erase}(\hat{\tau}_2), \varepsilon) <: \hat{\tau}_2$ and $\hat{\tau}_1 <: \texttt{annot}(\texttt{erase}(\hat{\tau}_1), \varepsilon)$, because the result will hold by S-EFFECTS. To achieve this we shall inductively apply **lemma 2** to $\hat{\tau}_2$ and **lemma 1** to $\hat{\tau}_1$.

From $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ we have $\texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2) \subseteq \varepsilon$ and therefore $\texttt{ho-effects}(\hat{\tau}_2) \subseteq \varepsilon$. From $\texttt{safe}(\hat{\tau}, \varepsilon)$ we have $\texttt{safe}(\hat{\tau}_2, \varepsilon)$. Therefore we can apply **lemma 2** to $\hat{\tau}_2$.

From $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ we have $\texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2) \subseteq \varepsilon$ and therefore $\texttt{effects}(\hat{\tau}_1) \subseteq \varepsilon$. From $\texttt{safe}(\hat{\tau}, \varepsilon)$ we have $\texttt{ho-safe}(\hat{\tau}_1, \varepsilon)$. Therefore we can apply **lemma 1** to $\hat{\tau}_1$.

□

Lemma 16 (CC Annotation)**.** *If the following are true:*

1. $\hat{\Gamma} \vdash \hat{v}_i : \hat{\tau}_i$ `with` $\varnothing$
2. $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e : \tau$
3. $\texttt{effects}(\hat{\tau}_i) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \cup \texttt{effects}(\texttt{annot}(\Gamma, \varnothing)) \subseteq \varepsilon_s$
4. $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s)$

Then $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)$ `with` ε_s.

Proof. By induction on the derivation of $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e : \tau$. When applying the inductive assumption, e, τ, and Γ may vary, but the other variables are fixed.

Case: T-VAR. Then $e = x$ and $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash x : \tau$. Either $x = y$ or $x \neq y$.

Subcase 1: $x = y$. Then $y : \texttt{erase}(\hat{\tau}_i) \vdash y : \tau$ so $\tau = \texttt{erase}(\hat{\tau}_i)$. By ε-VAR, $y : \hat{\tau}_i \vdash x : \hat{\tau}_i$ `with` $\varnothing$. By definition $\texttt{annot}(x, \varepsilon_s) = x$, so (5) $y : \hat{\tau}_i \vdash \texttt{annot}(x, \varepsilon_s) : \hat{\tau}_i$ `with` $\varnothing$. By (3) and (4) we know $\texttt{effects}(\hat{\tau}_i) \subseteq \varepsilon_s$ and $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s)$. By the approximation lemma, $\hat{\tau}_i <: \texttt{annot}(\texttt{erase}(\hat{\tau}_i), \varepsilon_s)$. We know $\texttt{erase}(\hat{\tau}_i) = \tau$, so this judgement can be rewritten as $\hat{\tau}_i <: \texttt{annot}(\tau, \varepsilon_s)$. From this we can use ε-SUBSUME to narrow the type of (5) and widen the approximate effects of (5) from $\varnothing$ to ε_s, giving $y : \hat{\tau}_i \vdash \texttt{annot}(x, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)$ `with` ε_s.

Finally, by widening the context, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), \hat{\tau}_i \vdash \texttt{annot}(x, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$.

Subcase 2: $x \neq y$. Because $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash x : \tau$ and $x \neq y$ then $x : \tau \in \Gamma$. Then $x : \texttt{annot}(\tau, \varepsilon_s) \in \texttt{annot}(\Gamma, \varepsilon_s)$ so $\texttt{annot}(\Gamma, \varepsilon_s) \vdash x : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varnothing$ by ε-VAR. By definition $\texttt{annot}(x, \varepsilon_s) = x$, so $\texttt{annot}(\Gamma, \varepsilon_s) \vdash \texttt{annot}(x, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varnothing$. Applying ε-SUBSUME gives $\texttt{annot}(\Gamma, \varepsilon_s) \vdash \texttt{annot}(x, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$. By widening the context $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon'$.

Case: T-RESOURCE. Then $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash r : \{r\}$. By ε-RESOURCE, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau}_i \vdash r : \{r\}\ \texttt{with}\ \varnothing$. Applying definitions, $\texttt{annot}(r, \varepsilon) = r$ and $\texttt{annot}(\{r\}, \varepsilon_s) = \{r\}$, so this judgement can be rewritten as $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varnothing$. By ε-SUBSUME, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$.

Case: T-ABS. Then $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash \lambda x : \tau_2.e_{body} : \tau_2 \to \tau_3$. Applying definitions, (5) $\texttt{annot}(e, \varepsilon_s) = \texttt{annot}(\lambda x : \tau_2.e_{body}, \varepsilon_s) = \lambda x : \texttt{annot}(\tau_2, \varepsilon_s).\texttt{annot}(e_{body}, \varepsilon_s)$ and $\texttt{annot}(\tau, \varepsilon_s) = \texttt{annot}(\tau_2 \to \tau_3, \varepsilon_s) = \texttt{annot}(\tau_2, \varepsilon_s) \to_{\varepsilon_s} \texttt{annot}(\tau_3, \varepsilon_s)$. By inversion on T-ABS, we get the sub-derivation (6) $\Gamma, y : \texttt{erase}(\hat{\tau}_i), x : \tau_2 \vdash e_{body} : \tau_2$. We shall apply the inductive assumption to this judgement with an unannotated context consisting of $\Gamma, x : \tau_2$. To be a valid application of the lemma, it is required that $\texttt{effects}(\texttt{annot}(\Gamma, x : \tau_2, \varnothing) \subseteq \varepsilon_s$. We already know $\texttt{effects}(\texttt{annot}(\Gamma, \varnothing)) \subseteq \varepsilon_s$ by assumption (3). Also by assumption (3), $\texttt{ho-effects}(\texttt{annot}(\tau_2 \to \tau_3, \varnothing)) \subseteq \varepsilon_s$; then by definition of $\texttt{ho-effects}$, $\texttt{effects}(\texttt{annot}(\tau_2, \varnothing)) \subseteq \texttt{ho-effects}(\texttt{annot}(\tau_2 \to \tau_3, \varnothing))$, so $\texttt{effects}(\texttt{annot}(x : \tau_2,))\varepsilon_s) \subseteq \varepsilon_s$ by transitivity. Then by applying the inductive assumption to (6), $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), \texttt{annot}(x : \tau_2, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e_{body}, \varepsilon_s) : \texttt{annot}(\tau_3, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$. By ε-ABS, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \lambda x : \texttt{annot}(\hat{\tau}_2, \varepsilon_s).\texttt{annot}(e_{body}, \varepsilon_s) : \texttt{annot}(\hat{\tau}_2, \varepsilon_s) \to_{\varepsilon_s} \texttt{annot}(\hat{\tau}_3, \varepsilon_s)\ \texttt{with}\ \varnothing$. By applying the identities from (5), this judgement can be rewritten as $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varnothing$. Finally, by applying ε-SUBSUME, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$.

Case: T-APP. Then $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e_1\ e_2 : \tau_3$ and by inversion $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e_1 : \tau_2 \to \tau_3$ and $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e_2 : \tau_2$. By applying the inductive assumption to these judgements, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e_1, \varepsilon_2) : \texttt{annot}(\tau_2, \varepsilon_s) \to_{\varepsilon_s} \texttt{annot}(\tau_3, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$ and $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau} \vdash \texttt{annot}(e_2, \varepsilon_s) : \texttt{annot}(\tau_2, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$. Then by ε-APP, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau} \vdash \texttt{annot}(e_1, \varepsilon_s)\ \texttt{annot}(e_2, \varepsilon_s) : \texttt{annot}(\tau_3, \varepsilon)\ \texttt{with}\ \varepsilon$. Unfolding the definition of $\texttt{annot}$, this judgement can be rewritten as $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau} \vdash \texttt{annot}(e_1\ e_2, \varepsilon_s) : \texttt{annot}(\tau_3, \varepsilon)\ \texttt{with}\ \varepsilon$. Finally, because $e = e_1\ e_2$ and $\tau = \tau_3$, this is the same as $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon$.

Case: T-OPERCALL. Then $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e_1.\pi : \texttt{Unit}$. By inversion we get the sub-derivation $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e_1 : \{\bar{r}\}$. Applying the inductive assumption, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau}_i \vdash \texttt{annot}(e_1, \varepsilon_s) : \texttt{annot}(\{\bar{r}\}, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$. By definition, $\texttt{annot}(\{\bar{r}\}, \varepsilon_s) = \{\bar{r}\}$, so this judgement can be rewritten as $\hat{\Gamma}, \texttt{annot}(\Gamma, \varnothing), y : \hat{\tau}_i \vdash e_1 : \{\bar{r}\}\ \texttt{with}\ \varepsilon_s$. By ε-OPERCALL, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varnothing), y : \hat{\tau} \vdash \texttt{annot}(e_1.\pi, \varepsilon_s) : \{\bar{r}\}\ \texttt{with}\ \varepsilon_s \cup \{\bar{r}.\pi\}$. All that remains is to show $\{\bar{r}.\pi\} \subseteq \varepsilon$. We shall do this by considering which subcontext left of the turnstile is capturing $\{\bar{r}\}$. Technically, $\hat{\Gamma}$ may not have a binding for every $r \in \bar{r}$: the judgement for e_1 might be derived using S-RESOURCES and ε-SUBSUME. However, at least one binding for some $r \in \bar{r}$ must be present in $\hat{\Gamma}$ to get the original typing judgement being subsumed, so we shall assume without loss of generality that $\hat{\Gamma}$ contains a binding for every $r \in \bar{r}$.

Subcase 1: $\{\bar{r}\} = \hat{\tau}$. By assumption (3), $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s$, so $\bar{r}.\pi \subseteq \{r.\pi \mid r \in \bar{r}, \pi \in$

$\Pi\} = \texttt{effects}(\{\bar{r}\}) \subseteq \varepsilon_s$.

Subcase 2: $r : \{\bar{r}\} \in \texttt{annot}(\Gamma, \varepsilon_s)$. Then $\bar{r}.\pi \in \texttt{effects}(\{\bar{r}\}) \subseteq \texttt{effects}(\texttt{annot}(\Gamma, \varnothing))$, and by assumption (3) $\texttt{effects}(\texttt{annot}(\Gamma, \varnothing)) \subseteq \varepsilon_s$, so $\bar{r}.\pi \in \varepsilon_s$.

Subcase 3: $r : \{\bar{r}\} \in \hat{\Gamma}$. Because $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1 : \{\bar{r}\}$, then $\bar{r} \in \Gamma$ or $r = \tau$. If $r \in \texttt{annot}(\Gamma, \varnothing)$ then subcase 2 holds. Else $r = \texttt{erase}(\hat{\tau})$. Because $\hat{\tau} = \{\bar{r}\}$, then $\texttt{erase}(\{\bar{r}\}) = \{\bar{r}\}$, so $\hat{\tau} = \tau$; therefore subcase 1 holds. □

Theorem 22 (CC Preservation)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$*, then* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$*, where* $\hat{e}_B <: \hat{e}_A$ *and* $\varepsilon \cup \varepsilon_B \subseteq \varepsilon_A$*, for some* $\hat{e}_B, \varepsilon, \hat{\tau}_B, \varepsilon_B$*.*

Proof. By induction on the derivation of $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ and then the derivation of $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$.

Case: ε-IMPORT. Then by inversion on the rules used, the following are true:

1. $\hat{e}_A = \texttt{import}(\varepsilon_s)\ x = \hat{v}_i \texttt{ in } e$
2. $x : \texttt{erase}(\hat{\tau}_i) \vdash e : \tau$
3. $\hat{\Gamma} \vdash \hat{e}_i : \hat{\tau}_i \texttt{ with } \varepsilon_1$
4. $\hat{\Gamma} \vdash \hat{e}_A : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s \cup \varepsilon_1$
5. $\texttt{effects}(\hat{\tau}_i) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s$
6. $\texttt{ho-safe}(\hat{\tau}_i, \varepsilon_s)$

Subcase 1: E-IMPORT1. Then $\texttt{import}(\varepsilon_s)\ x = \hat{e}_i \texttt{ in } e \longrightarrow \texttt{import}(\varepsilon_s)\ x = \hat{e}'_i \texttt{ in } e \mid \varepsilon$ and by inversion, $\hat{e}_i \longrightarrow \hat{e}'_i \mid \varepsilon$. By inductive assumption and subsumption, $\hat{\Gamma} \vdash \hat{e}'_i : \hat{\tau}'_i \texttt{ with } \varepsilon_1$. Then by ε-IMPORT, $\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}'_i \texttt{ in } e : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s$.

Subcase 2: E-IMPORT2. Then $\hat{e}_i = \hat{v}_i$ is a value and $\varepsilon_1 = \varnothing$ by canonical forms. Apply the annotation lemma with $\Gamma = \varnothing$ to get $\hat{\Gamma}, x : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s$. From assumption (4) and canonical forms we have $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}_i \texttt{ with } \varnothing$. Applying the substitution lemma, $\hat{\Gamma} \vdash [\hat{v}_i / x]\texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s$. Then $\varepsilon \cup \varepsilon_B = \varepsilon_A = \varepsilon_s$ and $\tau_A = \tau_B = \texttt{annot}(\tau, \varepsilon_s)$.

□

Theorem 23 (CC Single-step Soundness)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A$ *is not a value, then* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$*, where* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $\hat{e}_B$*,* ε*,* $\hat{\tau}_B$*, and* ε_B*.*

Theorem 24 (CC Multi-step Soundness)**.** *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A \longrightarrow^* e_B \mid \varepsilon$*, then* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$*, where* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $\hat{\tau}_B$*,* ε_B*.*

Proof. The same as for OC. □

Bibliography

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] BIRD, C., GALL, H., MURPHY, B., AND DEVANBU, P. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox.

[3] BRACHA, G., VON DER AHÉ, P., BYKOV, V., KASHAI, Y., MADDOX, W., AND MIRANDA, E. Modules as Objects in Newspeak. In *European Conference on Object-Oriented Programming* (2010).

[4] CHEN, S., ROSS, D., AND WANG, Y.-M. An Analysis of Browser Domain-isolation Bugs and a Light-weight Transparent Defense Mechanism. In *Conference on Computer and Communications Security* (2007).

[5] CHURCH, A. A formulation of the simple theory of types. *American Journal of Mathematics 5* (1940), 56–68.

[6] COKER, Z., MAASS, M., DING, T., LE GOUES, C., AND SUNSHINE, J. Evaluating the Flexibility of the Java Sandbox. In *Annual Computer Security Applications Conference* (2015).

[7] DENNIS, J. B., AND VAN HORN, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM 9*, 3 (1966), 143–155.

[8] DEVRIESE, D., BIRKEDAL, L., AND PIESSENS, F. Ieee european symposium on security and privacy. In *Reasoning about Object Capabilities with Logical Relations and Effect Parametricity* (2016).

[9] DROSSOPOULOU, S., NOBLE, J., MILLER, M. S., AND MURRAY, T. Reasoning about risk and trust in an open word. In *European Conference on Object-Oriented Programming* (2007), pp. 451–475.

[10] KNEUPER, R. Limits of formal methods. *Formal Aspects of Computing 3*, 1 (1997).

[11] KURILOVA, D., POTANIN, A., AND ALDRICH, J. Modules in wyvern: Advanced control over security and privacy. In *Proceedings of the Symposium and Bootcamp on the Science of Security* (New York, NY, USA, 2016), HotSos '16, ACM, pp. 68–68.

[12] LIU, F. A study of capability-based effect systems. Master's thesis, École Polytechnique Fédérale de Lausanne, 2016.

[13] LUCASSEN, J. M., AND GIFFORD, D. K. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 47–57.

[14] MAASS, M. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016.

[15] MAFFEIS, S., MITCHELL, J. C., AND TALY, A. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy* (2010).

[16] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[17] NIELSON, F., AND NELSON, H. R. Type and Effect Systems. pp. 114–136.

[18] NISTOR, L., KURILOVA, D., BALZER, S., CHUNG, B., POTANIN, A., AND ALDRICH, J. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and inHerItance* (New York, NY, USA, 2013), MASPEGHI '13, ACM, pp. 9–16.

[19] ODERSKY, M., ALTHERR, P., CREMET, V., DUBOCHET, G., EMIR, B., HALLER, P., MICHELOUD, S., MIHAYLOV, N., MOORS, A., RYTZ, L., SCHINZ, M., STENMAN, E., AND ZENGER, M. Scala Language Specification. `http://scala-lang.org/files/archive/spec/2.11/`. Last accessed: Nov 2016.

[20] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, Cambridge, MA, USA, 2002.

[21] RYTZ, L., ODERSKY, M., AND HALLER, P. Lightweight polymorphic effects. In *ECOOP* (2012).

[22] SALTZER, J. H. Protection and the Control of Information Sharing in Multics. *Communications of the ACM 17*, 7 (1974), 388–402.

[23] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. In *Proceedings of the IEEE 63-9* (1975).

[24] SCHREUDERS, Z. C., MCGILL, T., AND PAYNE, C. The State of the Art of Application Restrictions and Sandboxes: A Survey of Application-oriented Access Controls and Their Shortfalls. *Computers and Security 32* (2013), 219–241.

[25] TALPIN, J.-P., AND JOUVELOT, P. The type and effect discipline. *Information and Computation 111*, 2 (1994), 245–296.

[26] TANG, Y.-M. *Control-Flow Analysis by Effect Systems and Abstract Interpretation*. PhD thesis, Ecole des Mines de Paris, 1994.

[27] TER LOUW, M., BISHT, P., AND VENKATAKRISHNAN, V. Analysis of Hypertext Isolation Techniques for XSS Prevention. *Web 2.0 Security and Privacy* (2008).

[28] WATSON, R. N. M. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *USENIX Workshop on Offensive Technologies* (2007).