Capability-Flavoured Effects

by

Aaron Craig

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Bachelor of Science with Honours
in Computer Science.

Victoria University of Wellington
2017

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 which can help developers make more informed decisions about whether to trust code. To demonstrate how this works a capability calculus `CC` is developed and proven sound.

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 design 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 any development environment adhering to ideas of *code ownership*, wherein groups of developers function as experts over certain components in the system. When one group's components must interact with another's, they can make false assumptions or violate their internal constraints by using them incorrectly. 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 plug-ins, in which case third-party code could be written by anyone, including the untrustworthy. One kind of application which does this is the web mash-up, which brings together several existing, disparate services into one system. In both cases we want the entire system to function securely, despite the existence of untrustworthy components.

It is difficult to determine if a piece of code is trustworthy, but a range of techniques might be used. One approach is to *sandbox* the untrusted code inside a virtual environment. If anything goes wrong, damage is theoretically limited to the virtual environment, but in practice, this approach has many vulnerabilities [5, 12, 26, 22]. On the other hand, verification techniques allow for a robust analysis of the behaviour of code, but are heavyweight and require the developers using them to have a deep understanding of the techniques being employed [8]. Furthermore, verification requires one to supply a complete specification of the system, which may itself be an undefined or evolving artifact during the development process. Lightweight analyses, such as type systems, are easy for the developer to use, but existing languages lack adequate controls for detecting and isolating untrustworthy components [3, 25]. A qualitative approach might instead be employed, 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 [20]. For example, a logger module, which needs only to append to a file, should not have 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 [21]. This report is interested in the class of lightweight analyses, and in particular how type systems could be used to reject unsafe programs or put developers in a more informed position to make qualitative assessments about their code.

One approach to privilege separation is the capability model. A *capability* is an unforgeable token granting its bearer permission to perform some operation [6]. 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 [15]. For example, a logger might `append` to a `File`, and so executing its code would incur the `File.append` effect. In the capability model, this would require the logger to possess a capability granting it the ability to append to files.

Although the idea of a capability is an old one in the access literature, there has been recent interest in the application of the idea to programming language design. Miller has identified the ways in which capabilities should proliferate to encourage *robust composition* — a set of ideas summarised as "only connectivity begets connectivity" [14]. In this paradigm, actors in a program are explicitly parametrised by what capabilities they use. This enables one to reason about what privileges a component might exercise by examining its interface. Building on these ideas, Maffeis et. al. formalised the notion of a *capability-safe* language, showing a subset of Caja (a JavaScript implementation) is capability-safe [13].

Effect systems were introduced by Lucassen and Gifford for the purposes of optimising pure code [11]. They have also been applied to problems such as determining which functions might be invoked in a program [24], or determining which regions in memory may be accessed or updated [23]. 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 writes 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 verbosity is the main reason [19]. Successive works have focussed on reducing the developer overhead through techniques such as effect-inference, but the benefit of capabilities for enabling effect-inference has not received much attention. Because capabilities encapsulate the source of effects, and because capability-safety impose constraints on how they propagate through a system, the task of determining what effects might be incurred by a piece of code is simplified. This is the key contribution of this report: the idea that capability-safety facilitates a low-cost effect system with minimal user overhead.

We begin this report 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, effect-annotated lambda calculus with a simple notion of capabilities and effect. Dropping the requirement that all code in a program must be effect-annotated, we develop the Capability Calculus `CC`, which permits the nesting of unannotated code inside annotated code in a controlled, capability-safe manner with a new `import` construct. A safe inference about the unannotated code can be made at these junctions. In chapter 4 we demonstrate how `CC` can model practical examples, finishing with a summary and comparison of some of the existing work in this area.

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 defined, and how to prove the type system is correct. For this purpose, we present a toy language called EBL. We then summarise a variant of the simply-typed lambda calculus $\lambda^{\rightarrow}$. $\lambda^{\rightarrow}$ is an historically important model of computation which serves as a 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 how it is executed; and static rules, which determine whether programs meet certain well-behavedness properties. When a language has been defined we want to know its static rules are mathematically correct with respect to the dynamic rules.

Alongside the explanation of these concepts we develop EBL, 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. Although EBL is not very interesting, it will illustrate the general approach in 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 either a variable, 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, $3+2$, $\texttt{false} \vee \texttt{true}$, $3 \vee \texttt{false}$, $\texttt{true} + \texttt{false}$, $\texttt{let}\ x = 3\ \texttt{in}\ x + 1$.

Although the grammar hs no brackets, a string like $3 + (x + 2)$ should be seen as a short-hand for the corresponding abstract syntax tree (AST), whose structure is given by the rules of the grammar. For some strings the AST is ambiguous, as in $3 + x + 2$, which might be parsed as $3 + (x + 2)$ or as $(3 + x) + 2$. How we parse and disambiguate strings is not relevant to us, so throughout the report we only ever consider strings which unambiguously correspond to terms in the grammar.

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

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 explaining how it is executed. This is given as a set of *inference rules*, which are given as a set of premises above a dividing line. If the premises above the line hold, they imply the result below the line. The results are called *judgements*. If an inference rule has no premises it is an *axiom*. A particular application of an inference rule is a *derivation*. Figure **??** gives the dynamic rules for `EBL`, which specify a binary relation $\longrightarrow$, representing a single computational step. When the relation holds of a particular pair, we say the judgement $e \longrightarrow e'$ holds, and that e reduces to e'.

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

$\boxed{e \longrightarrow e}$

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

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

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

Figure 2.2: Inference rules for single-step reductions.

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. The notation for this is $[v_1/x]e_2$. For example, $\texttt{let}\ x = 1\ \texttt{in}\ x + 1$ reduces to $1 + 1$ by E-LET2.

Formally, substitution is a function operating 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.

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

$[e'/y]l = l$
$[e'/y]b = b$
$[e'/y]x = v$, 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, changing them will not change the semantics of the program. Therefore, we freely and implicitly interchange expressions which are equivalent up to the naming of bound variables. This process is called α-conversion [18, p. 71]. Consequently, we assume variables are (re-)named in this way to avoid these problems and to play nicely with the definition of `substitution`.

Lastly, note how in an expression like $\texttt{let}\ x = 1 + 1\ \texttt{in}\ x + 1$. According to the rules, $1 + 1$ would first be reduced to 2 before the substitution is made on $x + 1$. This strategy of reducing expressions to irreducible forms before they are bound to their names is known as *call-by-value*. Some languages — such as Haskell — are not call-by-value, but we shall only consider languages with call-by-value semantics.

From the single-step reduction relation, we define a 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.

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

$$\frac{}{e \longrightarrow^* e}\ (\text{E-MultiStep1}) \qquad \frac{e \longrightarrow e'}{e \longrightarrow^* e'}\ (\text{E-MultiStep2})$$

$$\frac{e \longrightarrow^* e' \quad e' \longrightarrow^* e''}{e \longrightarrow^* e''}\ (\text{E-MultiStep3})$$

Figure 2.4: Dynamic rules.

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

[1] We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation [18, p. 39].

stuck because $+$ is an operation on numbers, and `false` is a boolean. Another example is $\texttt{x} + \texttt{1}$, 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. Another says that every variable in a program must be declared before it is used. If a program satisfies these well-behavedness properties, its execution will never get stuck or produce a nonsense answer. We also want to know if a program satisfies these properties before it is executed.

To achieve this we add static rules, enriching `EBL` with a basic type system, which associates each expression with a type. 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 constructrs for the type system are given as a grammar in Figure 2.5. There are two types: `Nat` and `Bool`, and a notion of a typing context, which map 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.

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

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 $\varnothing \vdash e : \tau$.

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

$$
\frac{}{\Gamma, x : \texttt{Int} \vdash x : \texttt{Int}}\ (\text{T-}\textsc{Var})
\qquad
\frac{}{\vdash b : \texttt{Bool}}\ (\text{T-}\textsc{Bool})
\qquad
\frac{}{\vdash l : \texttt{Nat}}\ (\text{T-}\textsc{Nat})
$$

$$
\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \texttt{Bool}}{\Gamma \vdash e_1 \vee e_2 : \texttt{Bool}}\ (\text{T-}\textsc{Or})
\qquad
\frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 + e_2 : \texttt{Nat}}\ (\text{T-}\textsc{Add})
$$

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

Figure 2.6: Inference rules for typing arithmetic expressions.

T-BOOL and T-NAT are rules which say that constants always type to `Bool` or `Nat`. T-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 typecheck the body of the `let` expression. This lets $\texttt{let}\ x = 1\ \texttt{in}\ x + 1$ typecheck, 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.

$$\dfrac{\dfrac{}{\vdash 1 : \texttt{Nat}}\ (\textsc{T-Nat}) \qquad \dfrac{\dfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}\ (\textsc{T-Var}) \qquad \dfrac{}{x : \texttt{Int} \vdash 1 : \texttt{Int}}\ (\textsc{T-Nat})}{x : \texttt{Int} \vdash x + 1 : \texttt{Int}}\ (\textsc{T-Add})}{\vdash \texttt{let}\ x = 1\ \texttt{in}\ x + 1 : \texttt{Int}}\ (\textsc{T-Let})$$

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 as a mapping from variables to types. $x : \texttt{Int}, y : \texttt{Int}$ is really the same thing as $y : \texttt{Int}, x : \texttt{Int}$. Furthermore, if 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 $x : \texttt{Int}, y : \texttt{Int} \vdash x : \texttt{Int}$. We can ensure these properties with the rules in Figure 2.8.

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

$$\dfrac{\Gamma \vdash e : \tau \quad \Gamma' \textit{ is a permutation of } \Gamma}{\Gamma' \vdash e : \tau}\ (\Gamma\textsc{-Permute}) \qquad \dfrac{\Gamma \vdash e : \tau \quad x \notin \texttt{dom}(\Gamma)}{\Gamma, x : \tau' \vdash e : \tau}\ (\Gamma\textsc{-Widen})$$

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. Another property we desire of Γ is that it contains no duplicate variables. However, by the convention of α-renaming, all programs have unique variable names, so no rule is required.

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

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

Figure 2.9: Definition of `dom`.

These rules cause typing contexts to behave as we expect, but in practice the notation for contexts and how to manipulate are so conventional that we shall not bother to mention them again. The rules above will be applied automatically and left out of derivation trees.

It is worth mentioning that most languages have a *subtyping* judgement, written $\tau_1 <: \tau_2$, meaning expressions of type τ_1 may be provided anywhere in a program where an expression of type τ_2 are expected, and the program will still be well-typed. EBL has no subtyping rules, but we shall encounter some later.

2.1.4 Soundness

Having defined the static rules of EBL we can try to apply the rules to those examples in the last section which got stuck during reduction or evaluated to some nonsense result, but there is no application of rules that will ascribe a type to these examples, signalling that these do not meet our well-behavedness properties. However, we want to know these rules are correct in that they reject every program which goes wrong during execution. This property is called *soundness*, and asserts that the static rules are correct with respect to the dynamic rules. 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* $\vdash e' : \tau$ *for some* e'*.*

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.

By combining progress and preservation, we know that a runtime type-error can never occur as the result of a single-step reduction. This is soundness for small-step reductions. Once this has been established, we may extend this to multi-step reductions by inducting on the length of the multi-step and appealing to the soundness of single-step reductions, which yields the following theorem.

Theorem 3 (EBL Soundness)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow^* e'$ *then* $\vdash e' : \tau$*.*

All these theorems are proven by structural induction on the typing rule $\Gamma \vdash e : \tau$ used in the premise and, where appropriate, on the reduction rule $e \longrightarrow e'$ used.

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 typing rules. The second is the substitution lemma, which says if a term is well-typed in a context $\Gamma, x : \tau' \vdash e : \tau$, and you replace variable x with an expression e' of type τ, then $\Gamma \vdash [e'/x]e : \tau$. Note how e and $[e'/x]$ are ascribed the same type in the same context. In `EBL`, this lemma is needed to show that the reduction step in E-LET2 is type-preserving.

Precise formulations of these lemmas for `EBL` is 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* $\Gamma \vdash [e'/x]e : \tau$.

To summarise, soundness establishes that the static rules of a language are correct with respect to its semantics. The converse of soundness is also interesting to consider: if a program has no runtime type error, will the type system accept it? This is called *completeness*. Few type systems are complete, including `EBL`. This means `EBL` might reject type safe programs. To show why, consider the Java program in Figure 2.10. This program is type-safe, because the only branch of the conditional which ever executes is the one which returns an `int`. However, Java will reject this program because, in general, statically determining which branches can or cannot execute is undecidable.

```java
public int doubleNum(int x) {
  if (true) return x + x;
  else return true;
}
```

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

This report is only ever concerned with proving soundness, but it is impotrant to recognise that being incomplete makes a type system inherently *conservative*, meaning it can reject type-safe programs or make over-estimations as to what will happen. 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" [18, p. 2]. In order to approximate, simplifying assumptions must be made, and these simplifying assumptions are what make the type-system sound; but assumptions which are too generalising may result in more and more type safe examples getting rejected.

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 [4], 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

$$
\begin{array}{lllr}
e & ::= & & \mathit{exprs:} \\
 & | & x & \mathit{variable} \\
 & | & e\ e & \mathit{application} \\
 & | & v & \mathit{value}
\end{array}
\qquad
\begin{array}{lllr}
v & ::= & & \mathit{values:} \\
 & | & \lambda x:\tau.e & \mathit{abstraction} \\
 & | & n & \texttt{Nat}\ \mathit{constant} \\
 & | & i & \texttt{Int}\ \mathit{constant}
\end{array}
$$

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

with natural and integer numbers, so we can familiarise ourselves with subtyping. A grammar for $\lambda^{\rightarrow}$ programs is given in Figure 2.11.

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 n, or the function literal $\lambda x : \tau.e$. To distinguish `Nat` constants from 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 our notation for distinguishing between the two categories. In the function literal $\lambda x : \tau.e$, e is the function body, x is the name of the argument to the function, and τ is the type of the argument. An example is $\lambda x : \texttt{Int}.\ x$, which 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`. The arrow $\rightarrow$ is a type constructor: it can be used to build 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. For example, 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 $(\texttt{Nat} \rightarrow \texttt{Nat}) \rightarrow (\texttt{Nat} \rightarrow \texttt{Int})$.

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

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

Before giving the small-step semantics, we need to define `substitution`. Its definition is given 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.

The dynamic rules for $\lambda^{\rightarrow}$ are summarised in Figure 2.14. The only reducible expression is a function application. E-APP1 will reduce the left-side of an application. If the left-side is a value, but the right-side is an expression, then E-APP2 will reduce the right-side. If the left side is a function and the right-side is a value, then the right-side is bound 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.

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

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

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

$\boxed{e \longrightarrow e}$

$$\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1' e_2 \mid \varepsilon}\ (\text{E-A}\textsc{pp}1) \qquad \frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1 e_2 \longrightarrow v_1 e_2' \mid \varepsilon}\ (\text{E-A}\textsc{pp}2)$$

$$\frac{}{(\lambda x : \tau.e)v_2 \longrightarrow [v_2/x]e \mid \varnothing}\ (\text{E-A}\textsc{pp}3)$$

Figure 2.14: Dynamic rules for $\lambda^{\rightarrow}$.

As with EBL, some expressions in $\lambda^{\rightarrow}$ exhibit strange behaviours due to type errors or undefined variables. For example, consider $e = (\lambda x : \texttt{Int}.\ x)(\lambda x : \texttt{Int}.\ x)$. Then $e \longrightarrow e$ by E-App3. This expression can be endlessly reduced! But intuitively, we want to exclude it as a well-behaved program, 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 excluded because y is undefined. To determine whether a program is well-behaved we can apply the static rules for $\lambda^{\rightarrow}$, summarised in Figure 2.16.

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

$$\frac{}{\Gamma \vdash n : \texttt{Nat}}\ (\text{T-N}\textsc{at}) \qquad \frac{}{\Gamma \vdash i : \texttt{Int}}\ (\text{T-I}\textsc{nt}) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{T-V}\textsc{ar})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \rightarrow \tau_2}\ (\text{T-A}\textsc{bs}) \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_3 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_3}\ (\text{T-A}\textsc{pp})$$

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

The first two rules state that a natural number constant can always be typed to Nat, and an integer cosntant 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 $\tau_2 \rightarrow \tau_3$ and the right-hand expression has type τ_2. The examples above will now reject: $(\lambda x : \texttt{Int}.\ x)\ (\lambda x : \texttt{Int}.\ x)$ does not type because $\vdash \lambda x : \texttt{Int}.\ x : \texttt{Int} \rightarrow \texttt{Int}$, 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 say τ_1 is a more specific type than τ_2, and that τ_2 is a more general type than τ_1. Subtyping judgements for $\lambda^{\rightarrow}$ are given in Figure **??**.

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

$$\frac{}{\tau <: \tau}\ (\text{S-R}\textsc{eflexive}) \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ (\text{S-T}\textsc{ransitive})$$

$$\frac{}{\texttt{Nat} <: \texttt{Int}}\ (\text{S-N}\textsc{at}) \qquad \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}\ (\text{S-A}\textsc{rrow})$$

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

The rules S-REFLEXIVE and S-TRANSITIVE make subtyping a pre-ordering relation on types. 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. The former is called *contravariance* and the latter *covariance*. We say functions are contravariant in their input type and covariant in their output type.

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

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.

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

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{T-Subsume}$$

Figure 2.17: The subsumption rule.

$$\cfrac{\cfrac{\cfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}\ (\textsc{T-Var})}{\vdash \lambda x : \texttt{Int}\ x : \texttt{Int} \to \texttt{Int}}\ (\textsc{T-Abs}) \qquad \cfrac{\cfrac{}{\vdash 3_{\mathbb{N}} : \texttt{Nat}}\ (\textsc{T-Nat}) \qquad \cfrac{}{\texttt{Nat} <: \texttt{Int}}\ (\textsc{S-Nat})}{\vdash 3_{\mathbb{N}} : \texttt{Int}}\ (\textsc{T-Subsume})}{(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} : \texttt{Int}}\ (\textsc{T-App})$$

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

The definition of soundness for $\lambda^{\rightarrow}$ is very similar to EBL, but in the presence of subtyping, the type after reduction may get more specific than the type before reduction. To illustrate why this might happen, consider $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}}$. Figure 2.18 derives the judgement $\vdash (\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} : \texttt{Int}$. By E-App3, $(\lambda x : \texttt{Int}.\ x)\ 3_{\mathbb{N}} \longrightarrow 3_{\mathbb{N}}$. Then by T-Nat, $\vdash 3_{\mathbb{N}} : \texttt{Nat}$ — and $\texttt{Nat} <: \texttt{Int}$, 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. In general, we cannot statically determine the most precise type of an expression.

The soundness property for $\lambda^{\rightarrow}$ is given below. Note how $\tau_B <: \tau_A$, whereas in EBL, which had no subtyping, $\tau_B = \tau_A$.

Theorem 4 ($\lambda^{\rightarrow}$ Soundness). *If $\Gamma \vdash e_A : \tau_A$ and $e_A \longrightarrow^{*} e_B$, then $\Gamma \vdash e_B : \tau_B$, where $\tau_B <: \tau_A$.*

As a short aside, $\lambda^{\rightarrow}$ (and EBL) are *Turing-incomplete*, meaning there are programs which can be written in general-purpose languages that cannot be written in $\lambda^{\rightarrow}$. There are several routine ways to make $\lambda^{\rightarrow}$ as expressive as these 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, relative to 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 ascribe both a type and an effect to a piece of code; the effect component describes intensional information about the way in which a program executes [15]. To illustrate, we present a simplified version of SEA (Side-Effect Analysis), which is a calculus for reasoning about the set of memory cells that are written or read during execution [15]. Properly defining the small-step semantics of SEA requires us to cover more concepts which are largely irrelevant for the rest of the report, so we instead give a quick explanation of how they work.

2.3.1 SEA: Side-Effect Analysis

SEA is a lambda calculus with a type-and-effect system for reasoning about what memory cells are affected by computations. It extends $\lambda^{\rightarrow}$ with imperative constructs for creating, accessing, and updating reference variables. Our interest is in determining which cells might be created, accessed, or updated by a piece of code; effects in SEA are therefore one of those three operations on a particular cell. A particular memory cell is denoted π. It can be thought of as drawn from a set of memory cell variables Π.

A full definition of SEA would include its dynamic rules and a formulation and proof of soundness. Our purpose is to demonstrate how static rules can be used to describe what effects take place during a program execution. To this end, we omit a proper treatment of soundness and reduction, instead giving a quick summary.

The grammar for SEA programs 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 in the body of e_2, with its value initialised to e_1, at location π. $!x$ is used to access the value of the reference x. $x := e$ updates the value of x with e.

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

Figure 2.19: Grammar for SEA expressions.

In SEA an effect ϕ is the creation, reading, or writing of a reference at 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 2.20.

The runtime has the notion of a *store*, which maps each reference to the value defined in its cell. The store also keeps track of the location at which a reference was created. It can be enlarged and updated during runtime by the creation, access, and updating of references, each of which incurs a runtime effect $\texttt{new}_\pi$, $!\pi$, or $\pi :=$ respectively. Both

$$\begin{array}{lllllll}
\phi & ::= & & effects: & & & \\
& | & \mathtt{new}_\pi & ref.\ creation & \Phi & ::= & sets\ of\ effects: \\
& | & !\pi & ref.\ access & & | & \{\bar{\phi}\} \\
& | & \pi := & ref.\ update & & &
\end{array}$$

Figure 2.20: Grammar for effects and regions of SEA.

reading and writing to a reference x will return the value of x. 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.

In our presentation, the base types of SEA are Nat and Bool. $\tau_1 \rightarrow_\Phi \tau_2$ 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 effect in Φ will happen during execution. There is also a new type constructor ref. $\mathtt{ref}(\tau, \rho)$ is the type of a reference defined in one of the regions in ρ, which points to a value of type τ. The grammar for types is given in 2.21.

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

Figure 2.21: Grammar for SEA types.

There is a single judgement in SEA, which has the form $\Gamma \vdash e : \tau\ \mathtt{with}\ \Phi$. 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 2.24.

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 $\lambda^{\rightarrow}$, 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 $\tau_1 \rightarrow_\Phi \tau_2$. 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 typing rules are for manipulating references. T-READ will type $!x$ to the type τ referenced by x. Its effects are statically approximated as the singleton $\{!\pi\}$, where π is the location of x in the typing 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

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

$$\frac{}{\Gamma \vdash b : \mathtt{Bool}\ \mathtt{with}\ \varnothing}\ (\textsc{T-Bool}) \qquad \frac{}{\Gamma \vdash n : \mathtt{Nat}\ \mathtt{with}\ \varnothing}\ (\textsc{T-Nat})$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau\ \mathtt{with}\ \varnothing}\ (\textsc{T-Var}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2\ \mathtt{with}\ \Phi}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow_{\Phi} \tau_2\ \mathtt{with}\ \varnothing}\ (\textsc{T-Abs})$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow_{\Phi_3} \tau_3\ \mathtt{with}\ \Phi_1 \quad \Gamma \vdash e_2 : \tau_2\ \mathtt{with}\ \Phi_2}{\Gamma \vdash e_1\ e_2 : \tau_3\ \mathtt{with}\ \Phi_1 \cup \Phi_2 \cup \Phi_3}\ (\textsc{T-App})$$

$$\frac{}{\Gamma, x : \mathtt{ref}(\tau, \pi) \vdash !x : \tau\ \mathtt{with}\ \{!\pi\}}\ (\textsc{T-Read})$$

$$\frac{\Gamma, x : \mathtt{ref}(\tau, \pi) \vdash e : \tau\ \mathtt{with}\ \Phi}{\Gamma, x : \mathtt{ref}(\tau, \pi) \vdash x := e : \tau\ \mathtt{with}\ \Phi \cup \{\pi :=\}}\ (\textsc{T-Write})$$

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

Figure 2.22: Static rules for SEA.

the expressino 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 $x : \mathtt{ref}(\tau_1, \pi)$ 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 $e = \mathtt{new}_{l_1} x = 3\ \mathtt{in}\ x := 5$, which initialises a reference at location l_1 with 3, and then updates it to 5 . This can be typed as $\vdash e : \mathtt{Nat}\ \mathtt{with}\ \{l_1 :=\}$; a derivation tree is given in Figure 2.23.

$$\dfrac{\dfrac{}{\vdash 3 : \mathtt{Nat}\ \mathtt{with}\ \varnothing}\ (\textsc{T-Nat}) \qquad \dfrac{\dfrac{}{x : \mathtt{ref}(\mathtt{Nat}, l_1) \vdash 5 : \mathtt{Nat}\ \mathtt{with}\ \varnothing}\ (\textsc{T-Nat})}{x : \mathtt{ref}(\mathtt{Nat}, l_1) \vdash x := 5 : \mathtt{Nat}\ \mathtt{with}\ \{l_1 :=\}}\ (\textsc{T-Write})}{\vdash \mathtt{new}_{l_1} x = 3\ \mathtt{in}\ x := 5 : \mathtt{Nat}\ \mathtt{with}\ \{l_1 :=\}}\ (\textsc{T-New})$$

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

Currently, the expressive power of SEA is so low that the approximations from the static rules give *exactly* those effects which will be incurred at runtime. In more complex languages the approximations will stop being tight upper-bounds. As an example of why, consider an extended version of SEA with conditional expressions. The conditional $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$ 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 **??**.

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 approximated as the effects incured by reducing the guard,

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

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

Figure 2.24: Static rules for SEA.

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.

2.4 The Capability Model

A *capability* is a unique, unforgeable reference, granting its bearer permission to perform some operation [6]. 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 [14]:

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." There are an initial set of primitive capabilities passed into the program at the beginning of execution by the system environment or virtual machine, which grant operations over *resources* in the system environment. For example, a File might grant operations on a particular file in the file system. Often we conflate the primitive capabilities and the system resources they grant access to, referring to both as resources. A capability is either a primitive capability, or a function or object which captures another capability. An example of a non-primitive capability would be a Logger which, possessing a particular File, presents a confined subset of operations on it.

These rules restrict how capabilities spread throughout a program, requiring components to be instantiated with the capabilities they request. As a result, the exercise of any authority is explicit. By contrast, the implicit exercise of authority is known as *ambient authority*. If a language disallows ambient authority and only proliferates capabilities in the above ways, it is called *capability-safe*. Figure 2.25 demonstrates one way in which authority can be implicitly exercised in Java: a malicious implementation of `List.add` attempts to overwrite the user's `.bashrc` file. `MyList` gains this capability by importing `java.io.File` and instantiating new instances of a capability for the user's `.bashrc` file. In a capability-safe language, `MyList` would have to be given the `.bashrc` file on start-up from the system environment directly, or by someone that already possesses it. Another way to exercise ambient authority is through global state: if a capability is stored inside a global variable then any component can acess and use its operations without having been explicitly given it. Therefore, capability-safe languages must disallow global state and unrestricted imports.

```java
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);
  }
}
```

```java
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.

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`; a necessity at odds with the circumstances that may surround untrusted code and code ownership.

Capability-safe languages usually have first-class modules, meaning objects and modules are treated in a uniform manner. Modules, like objects, must be instantiated, and can be given their capabilities at this point. They are also bound by the same proliferation rules constraining objects, so the constraints of the capability model are preserved across module boundaries. First-class modules are not exclusive to capability-safe languages: Scala has first class modules [17], but is not capability-safe. Within the capability-safe languages there is considerable variation in style: Smalltalk is a dynamically-typed capability-safe language with first-class modules [2]. Wyvern is a statically-typed capability-safe language [16] with first-class modules [9].

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 primitive capabilities and their operations. Every function is annotated with what effects it might incur. The static rules of `OC` ascribe a type-and-effect to programs, and 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 [16]. A more thorough discussion of how Wyvern programs might be translated into the calculi is given in Chapter 4.

3.1 `OC`: Operation Calculus

The operation calculus `OC` extends $\lambda^{\rightarrow}$ with primitive capabilities and their operations. A primitive capability encapsulates some system resource; a primitive capability `File` might provide some operations on a particular file in the file system. For convenience we often conflate the capability granting operations with the resource itself. An effect is a particular operation invoked on some resource. Every function-type in `OC` is annotated with what effects may be incurred during execution of the function body. The static rules of `OC` can inspect this information and ascribe a set of effects to a piece of code, giving a static approximation to the runtime effects.

When a component is annotated with the effects it might incur, an effect-system can determine when the allowed authority of a component is being violated. Consider the pair of modules in Figure 3.1: the `Logger` possesses a `File` capability and exposes a single function `log` which incurs the `File.write` effect. The `Client`, when passed a `Logger`, will invoke its `log` function.

```
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()
```

Figure 3.1: The implementation of `Client.run` exceeds its specified authority.

Following the principle of least authority, `Client.run` is annotated as incurring $\varnothing$ as its effects and `Logger.log` is annotated as incurring $\{$`File.write`$\}$. But as `Client.run` invokes `Logger.log`, its implementation violates its specified authority. By the end of this section, we will have developed rules for `OC` that reject implementations inconsistent with their specification.

3.1.1 `OC` Grammar

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

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

Figure 3.2: Grammar for `OC`.

A resource literal r is a variable drawn from a fixed set R. Resources cannot be created or destroyed at runtime. They model those initial capabilities passed into the program which endow operations upon system resources. A `File` and a `Socket` are examples of resource literals.

An operation call $e.\pi$ is the invocation of a primitive operation π invoked on the resource described by e. For example, we might invoke the `open` operation on a `File` resource, which would be the operation call `File.open`. Operations are drawn from a fixed set Π. Like resources, they cannot be created or destroyed at runtime.

An effect is a pair $(r, \pi) \in R \times \Pi$. Sets of effects are denoted ε; a rule for them 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. We have defined ε syntactically as a sequence of pairs from $R \times \Pi$, but instances of ε should be interpreted as a set. 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.

$$\begin{array}{lccl} \varepsilon & ::= & & effects: \\ & | & \{\overline{r.\pi}\} & effect\ set \end{array}$$

Figure 3.3: Grammar for effects in OC.

Because the static rules of OC are only interested in where effects are incurred, we have chosen not to model the semantics of particular operations. In practice, operations might take arguments of particular types and return a value of a particular type `File.write("msgToWrite")`, for example. We make a simplifying assumption that all operations are null-ary and return the same dummy value, and that all operations are defined on all resources.

3.1.2 OC Dynamic Rules

Before giving the dynamic rules we extend the definition of `substitution` to be defined on the extra forms. The extra cases are given in Figure 3.4. The function is defined the same on existing forms as in $\lambda^{\rightarrow}$, so we do not repeat them. We also make an extra restriction in OC that a variable may 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, which violates the preservation of effects. Because we only consider the call-by-value strategy, in which expressions are reduced to values before being bound to names, this restriction is no issue.

`substitution` :: e × v × v → e

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

Figure 3.4: Substitution function in OC.

During reduction an operation call may be evaluated. When this happens a runtime effect is said to have taken place. The form of the single-step reduction judgement is now $e \longrightarrow e \mid \varepsilon$ to reflect this fact; the resulting pair is the reduced expression, and the set of effects incurred as a result. In the case of single-step reduction, this is at most a single effect. Judgements for single-step reductions are summarised in Figure 3.5.

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

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

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

Figure 3.5: Single-step reductions in OC.

The first three rules are analogous to the rules from $\lambda^{\rightarrow}$. 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. `unit` is a derived form. It is the only value of its type, so is used to represent the absence of information. Because we choose not to model the semantics of operation calls, `unit` is a good dummy-value for operation calls to be returning.

From the single-step reductions we define the multi-step reductions in Figure 3.6. A multi-step reduction consists of zero or more single-steps. The resulting effect-set is the union of all the effect-sets produced by the intermediate single-steps.

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

$$\frac{}{e \rightarrow^* e \mid \varnothing}\ (\text{E-MultiStep1}) \qquad \frac{e \rightarrow e' \mid \varepsilon}{e \rightarrow^* e' \mid \varepsilon}\ (\text{E-MultiStep2})$$

$$\frac{e \rightarrow^* e' \mid \varepsilon_1 \quad e' \rightarrow^* e'' \mid \varepsilon_2}{e \rightarrow^* e'' \mid \varepsilon_1 \cup \varepsilon_2}\ (\text{E-MultiStep3})$$

Figure 3.6: Multi-step reductions in OC.

3.1.3 OC Static Rules

A grammar for the types of OC is given in Figure 3.7. Typing contexts are the same as in $\lambda^{\rightarrow}$. The base types are sets of resources, denoted $\{\bar{r}\}$. If an expression is associated with type $\{\bar{r}\}$, then evaluating e will reduce to one of the resource literals $r \in \bar{r}$ (assuming it terminates). Although $\{\bar{r}\}$ is syntactically defined as a sequence, it should be interpreted as a set; $\{\texttt{File}, \texttt{Socket}\}$ is the same type as $\{\texttt{Socket}, \texttt{File}\}$. There is a single type-constructor, $\to_{\varepsilon}$. If an expression is associated with type $\tau_1 \to_{\varepsilon} \tau_2$, then it is a function which takes a τ_1 as input, returns a τ_2 as output, and during execution incurs no more than those effects in ε. If an effect $r.\pi \in \varepsilon$, then it is not guaranteed that $r.\pi$ will occur during function execution; but if $r.\pi \notin \varepsilon$, then it cannot occur during function execution.

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

Figure 3.7: Grammar for types in OC.

The only way for code to gain authority over a capability is to be given that capability as a function argument. Because functions in OC must have their input types annotated with the effect-set they might incur on the arrow, we say that OC programs are annotated.

Given a program, we want to know what set of effects might be incurred when it is executed. For example, $(\lambda c : \{\texttt{File}, \texttt{Socket}\}.c.write)\texttt{File}$ incurs $\texttt{File.write}$ when executed. Judgements in the type system of OC therefore ascribe a type and a set of effects to a piece of code. The judgement form is $\Gamma \vdash e : \tau \;\texttt{with}\; \varepsilon$, which can be read as meaning that e will successively reduce to terms of type τ and incur no more effects than those in ε. Static rules for OC are given in Figure 3.9.

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

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

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

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

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

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

ε-Var approximates the runtime effects of a variable as $\varnothing$. ε-Resource does the same. 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 approximates the runtime effects of a function literal as $\varnothing$; although the ascribed type has an arrow with a set of effects, equivalent to the approximate effects of the function body.

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 obtained from its arrow-type.

ε-OPERCALL approximates an operation call as: 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 $\Gamma \vdash e : \{\texttt{File}, \texttt{Socket}\}\ \texttt{with}\ \varnothing$. Then e could evaluate to `File`, in which case the actual runtime effect is $\texttt{File}.\pi$, or it could evaluate to `Socket`, in which case the actual runtime effect is $\texttt{Socket}.\pi$. Determining which will actually 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 $\vdash \texttt{unit} : \texttt{Unit}\ \texttt{with}\ \varnothing$ 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 rules. It says that the type can be narrowed or the effect-set widened in a judgement to produce a new judgement. The subtyping rules are given in Figure 3.9.

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

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

Figure 3.9: Subtyping judgements of `OC`.

The first subtyping rule is S-ARROW, which is similar to the rule for subtyping functions in $\lambda^{\rightarrow}$. The only addition is that the effects of the subtype must be contained in the effects of the supertype, so that instances of the subtype can only incur effects the supertype's interface is expecting.

The other subtyping rule is S-RESOURCE, which says a subset of resource sis also a subtype. To justify this rule, consider $\{\bar{r}\} <: \{\bar{r}_2\}$. Any value with type $\{\bar{r}_1\}$ can reduce to any resource literal in $\bar{r}_1$, so to be compatible with type $\{\bar{r}_2\}$, 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 $e = (\lambda \texttt{f} : \{\texttt{File}, \texttt{Socket}\}.\texttt{f}.\texttt{write})\ \texttt{File}$. The judgement $\vdash e : \texttt{Unit}\ \texttt{with}\ \{\texttt{File}.\texttt{write}, \texttt{Socket}.\texttt{write}\}$ holds, which says that executing this piece of code might incur either of `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. Recall that `unit` is a derived form and `Unit` a derived type.

These rules can be used to determine if a piece of code is safe. For example, a function which uses a logger might be $\texttt{e} = \lambda \texttt{l} : \texttt{File} \rightarrow_{\texttt{File.append}} \texttt{Unit}.\ \texttt{l}\ \texttt{unit}$. Applying the rules to the logger implementation `l` gives an approximation to the effects it might incur. With

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \{\texttt{F},\texttt{S}\} \vdash f : \{\texttt{F},\texttt{S}\}}\ (\varepsilon\text{-}\textsc{Var})}{\texttt{f} : \{\texttt{F},\texttt{S}\} \vdash \texttt{f.w} : \texttt{Unit with}\ \{\texttt{F.w},\texttt{S.w}\}}\ (\varepsilon\text{-}\textsc{OperCall})}{\lambda\texttt{f} : \{\texttt{F},\texttt{S}\}.\texttt{f.w} : \{\texttt{F},\texttt{S}\} \to_{\texttt{F.w},\texttt{S.w}} \texttt{Unit with}\ \varnothing}\ (\varepsilon\text{-}\textsc{Abs}) \qquad \dfrac{}{\vdash \texttt{F} : \{\texttt{F}\}\ \texttt{with}\ \varnothing}\ (\varepsilon\text{-}\textsc{Resource})}{\vdash (\lambda\texttt{f} : \{\texttt{F},\texttt{S}\}.\texttt{f.write})\ \texttt{F} : \texttt{Unit with}\ \{\texttt{F.w},\texttt{S.w}\}}\ (\varepsilon\text{-}\textsc{App})$$

Figure 3.10: Derivation tree for $(\lambda\texttt{f} : \{\texttt{File},\texttt{Socket}\}.\ \texttt{f.write})\ \texttt{File}$.

that information, we can decide if it is safe to use that particular logger. For example, if $\texttt{l} = \lambda\texttt{f} : \{\texttt{File}\}.\ \texttt{f.read}$ then by ε-ABS, $\vdash\ \texttt{l} : \{\texttt{File}\} \to_{\texttt{File.read}} \texttt{Unit with}\ \varnothing$. We can see that applying this function will incur the File.read function, alerting us that this code might be maliciuos. Furthermore, e l will not typecheck, because e expects a function with File.append on the arrow.

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. Intuitively, 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 5 (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* $e_B, \varepsilon, \tau_B, \varepsilon_B$.

In this formulation, ε_A is an approximation to what e_A will do when executed. e_A reduces to e_B, incurring the effects in ε, and e_B can be typed in the same context Γ with the type τ_B and effect-approximation 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 no further information about ε_B is stipulated.

Our approach to proving that multi-step reduction is sound will be to inductively appeal to the soundness of single-step reductions. This is tricky under the given definition of Soundness because it only relates the runtime effects to the approximation of the runtime effects *before* reduction. There are no constraints on the runtime effects *after* reduction. To accommodate a proof of multi-step soundness, we need a stronger version of soundness which relates the approximated effects before reduction (ε_A) to the approximated effects after reduction (ε_B).

First consider how the type after reduction relates to the type before reduction. In λ-calculi, the type after reduction can be the same or more specific (i.e. $\tau_B <: \tau_A$) than the type before reduction, but never less specific. The idea is that as we reduce the expression we gain more information about its precise type. Similarly, we want to allow for the approximation to get more specific after a reduction. To illustrate why, 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 $\texttt{File.write}$ as $\{\texttt{File.write}\}$; 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 6 (OC Single-step Soundness). *If $\Gamma \vdash e_A : \tau_A$ $\texttt{with}$ ε_A and e_A is not a value, then $e_A \longrightarrow e_B \mid \varepsilon$, where $\Gamma \vdash e_B : \tau_B$ $\texttt{with}$ 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 separately. These in turn rely on canonical forms and the substitution lemma, modified for OC 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 rule used is ε-SUBSUME, the following are true:*

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

The first two observations state that a variable will always type with approximate effects $\varnothing$. 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 lambda. Progress follows from Canonical Forms.

Theorem 7 (OC Progress). *If $\Gamma \vdash e : \tau$ $\texttt{with}$ ε and e is not a value or variable, then $e \longrightarrow e' \mid \varepsilon$, for some e', ε.*

Proof. By induction on $\Gamma \vdash e : \tau$ $\texttt{with}$ 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 $\Gamma, x : \tau' \vdash e : \tau$ $\texttt{with}$ ε and $\Gamma \vdash v : \tau'$ $\texttt{with}$ $\varnothing$ then $\Gamma \vdash [v/x]e : \tau$ $\texttt{with}$ ε.*

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

With this lemma, we can prove the preservation theorem.

Theorem 8 (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 $e_1 : \tau_2 \rightarrow_\varepsilon \tau_3$ `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 9 (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 10 (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 — if we relax this requirement, can a type system say anything useful about pieces of unannotated code? There are practical reasons to permit unannotated code in an effect-conscious language. Previous effect systems have been criticised for their verbosity [19], which might disincline a programmer from bothering to use them. 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.

In general, reasoning about unannotated code is difficult because there are no constraints on what effects might be incurred. Figure 3.11 demonstrates the issue: `someMethod` takes a function f as input and executes it, but the effects of f depend on the 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`.

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

Figure 3.11: What effects can `someMethod` incur?

A capability-safe design can help us: if the capabilities exercised by the unannotated code are supplied by an annotated environment, then whatever effects they capture are a conservative upper-bound on what can happen in the unannotated code. To demonstrate, consider a developer who wants to decide whether the module in Figure 3.12 is trustworthy. The module is a `Logger`, possessing two capabilities `File` and `Socket`, and providing a single unannotated function `log`.

```
resource module Logger
require File
require Socket

def log(x: Unit): Unit
  ...
```

Figure 3.12: What effects can `someMethod` incur?

What effects will be incurred if `Logger.log` is invoked? One approach is to examine its source code, but this manual process is tedious and error-prone. In many real-world situations the source code may not be available. 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, a developer may decide this implementation cannot be trusted and choose not to use it.

The reasoning we employed only required us to examine the interface of the unannotated code for the capabilities that passed into it. To model this situation in `CC`, we add a new `import` expression selecting what effects ε the unannotated code may exercise. The static rules can check if the capabilities being imported violate ε. If the rules accept the code, then ε is a safe approximation of the unannotated code's effects; that capability-safe design enables this inference is the key result. The rest of this chapter is devoted to formalising the idea and proving it sound.

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.

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

Figure 3.13: Effect calculus.

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.

The new form `import` introduces a name x with annotated definition $\hat{e}$ into a body of unannotated code e. This should be understood as e importing the capability $\hat{e}$. ε is the

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

Figure 3.14: Effect calculus.

authority which e is allowed to exercise, so any resources and operation calls used in e must be declared in ε. `import` is the only way to nest unannotated code inside annotated code. It is not possible to nest annotated code inside unannotated code, because of the general difficulty of reasoning about what the unannotated code may do. We will not be interested in unannotated programs, unless they appear 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 executing them. Because our static rules focus on what can be said about unannotated code nested inside annotated code, we take this 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` will select its 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 in the definition will stop you annotating a program with something silly, so any use of `annot` must be justified.

$\texttt{annot} :: \mathtt{e} \times \boldsymbol{\varepsilon} \dashrightarrow \hat{\mathtt{e}}$

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

$\texttt{annot} :: \boldsymbol{\tau} \times \boldsymbol{\varepsilon} \dashrightarrow \hat{\boldsymbol{\tau}}$

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

$\texttt{annot} :: \Gamma \times \boldsymbol{\varepsilon} \dashrightarrow \hat{\Gamma}$

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

Figure 3.15: Definition of `annot`.

Finally, before giving the dynamic rules we must update the definition of `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 stipulate that variables can only be replaced with values, to prevent the introduction of arbitrary effects.

$\texttt{substitution} :: \hat{\mathtt{e}} \times \hat{\mathtt{v}} \times \hat{\mathtt{v}} \dashrightarrow \hat{\mathtt{e}}$

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

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 the authority ε; this is $\texttt{annot}(e, \varepsilon)$. The name of the capability x is then replaced with its definition; this is $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$. The single-step incurs no effects.

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

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

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

Figure 3.17: New single-step reductions in `CC`.

Multi-step reductions in CC are defined the same as in OC. For brevity, they are not restated.

3.2.3 CC Static Rules

Since a term might be annotated or unannotated, 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 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, 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.

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 3.19 which removes the annotations from a type. We then add $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$ as a premise.

$\texttt{erase} :: \hat{\boldsymbol{\tau}} \rightarrow \boldsymbol{\tau}$

$$\begin{array}{l}\texttt{erase}(\{\bar{r}\})\\ \texttt{erase}(\hat{\tau}_1 \rightarrow_{\varepsilon} \hat{\tau}_2) = \texttt{erase}(\hat{\tau}_1) \rightarrow \texttt{erase}(\hat{\tau}_2)\end{array}$$

Figure 3.19: Definitions of annot and erase.

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 the ascribed type would be $\texttt{annot}(\tau, \varepsilon)$: the type of the unannotated code, annotated with its selected authority ε. The approximate effects are $\varepsilon_1 \cup \varepsilon$; the former comes from reducing the imported capability — which happens before the (annotated) body of the import is executed — and the latter contains all the effects which the unannotated code is allowed to incur. The first version of ε-IMPORT is given in Figure 3.20.

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

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

Figure 3.20: 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. Consider $\hat{e}' = \texttt{import}(\varnothing)\ x = \texttt{File in x.write}$, which imports a File and writes to it, but declares its authority as $\varnothing$. According to ε-IMPORT1, $\vdash \hat{e}' : \texttt{Unit with}\ \varnothing$, but this is clearly wrong since $\hat{e}'$ writes to File. We need to constrain the imported capability to only capture effects in 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 3.21. 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 3.22.

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

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

Figure 3.21: A first definition of effects.

The counterexample which defeated ε-IMPORT1 is now rejected by ε-IMPORT2, but there are still issues: the annotations on one import can be broken by another import.

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ (\varepsilon\text{-}\textsc{Import2})$$

Figure 3.22: A second rule for type-and-effect checking `import` expressions.

To illustrate, consider Figure 3.23 where two[1] capabilities are imported. This program imports a function go which, when given a $\texttt{Unit} \rightarrow_{\varnothing} \texttt{Unit}$ function with no effects, will execute it. The other import is `File`. The unannotated code creates a $\texttt{Unit} \rightarrow \texttt{Unit}$ function which writes to `File` and passes it to go, which subsequently incurs `File.write`.

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

Figure 3.23: Permitting multiple imports will break ε-Import2.

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 file-writing function has type $\texttt{unit} \rightarrow \texttt{unit}$, the unannotated world accepts it as well-typed. The `import` selects $\{\texttt{File}.*\}$ as its authority, 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 $\texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}$ 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 $\texttt{Unit} \rightarrow_{\{\texttt{File}.*\}} \texttt{Unit}$, 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 $r.\pi \in \texttt{ho-effects}(\hat{\tau})$.

`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.

[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 imports are allowed.

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

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

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

$$\texttt{ho-effects}(\{\bar{r}\}) = \varnothing$$
$$\texttt{ho-effects}(\hat{\tau}_1 \rightarrow_\varepsilon \hat{\tau}_2) = \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$$

Figure 3.24: Effect functions.

Recall that functions are contravariant in their input type and covariant in their output type. Similarly, 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 unannotated code. The counterexample from Figure 3.23 is now rejected, because $\texttt{ho-effects}(\texttt{Unit} \rightarrow_\varnothing \texttt{Unit}) \rightarrow_\varnothing \texttt{Unit}) = \varnothing$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$, but this is *still* not sufficient! Consider $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2)$. 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 $\rightarrow_{\varepsilon'}$. We want a relation like $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$, but where $\subseteq$ distributes over the input and output type. Figure 3.25 defines exactly this: `safe` is a distributive version of $\varepsilon \subseteq \texttt{effects}(\hat{\tau})$ and `ho-safe` is a distributive version of $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau})$.

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 3.26. 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 the unannotated code might pass to it..

The premises so far restrict what authority can be selected by unannotated code, but what about authority passed as a function argument? Consider $\hat{e} = \texttt{import}(\varnothing)\ x = \texttt{unit in}\ \lambda\texttt{f} : \texttt{File}.\ \texttt{f.write}$. The unannotated code selects no capabilities and returns a function which, when given `File`, incurs `File.write`. This satisfies the premises in ε-IMPORT, but its annotated type is ε-IMPORT is $\{\texttt{File}\} \rightarrow_\varnothing \texttt{Unit}$ — not good!

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

$$\frac{\{r.\pi \mid r \in \bar{r}, \pi \in \Pi\} \subseteq \varepsilon}{} \ (\textsc{Safe-Resource})$$

$$\frac{\varepsilon \subseteq \varepsilon' \quad \texttt{ho-safe}(\hat{\tau}_1, \varepsilon) \quad \texttt{safe}(\hat{\tau}_2, \varepsilon)}{\texttt{safe}(\hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2, \varepsilon)} \ (\textsc{Safe-Arrow})$$

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

$$\frac{}{\texttt{ho-safe}(\{\bar{r}\}, \varepsilon)} \ (\text{HOS}\textsc{afe-Resource})$$

$$\frac{\texttt{safe}(\hat{\tau}_1, \varepsilon) \quad \texttt{ho-safe}(\hat{\tau}_2, \varepsilon)}{\texttt{ho-safe}(\hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2, \varepsilon)} \ (\text{HOS}\textsc{afe-Arrow})$$

Figure 3.25: Safety judgements in the epsilon calculus.

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

$$\frac{\begin{array}{c} \hat{\Gamma} \vdash \hat{e} : \hat{\tau} \ \texttt{with} \ \varepsilon_1 \qquad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon \\[1ex] \texttt{ho-safe}(\hat{\tau}, \varepsilon) \qquad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau \end{array}}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon) \ x = \hat{e} \ \texttt{in} \ e : \texttt{annot}(\tau, \varepsilon) \ \texttt{with} \ \varepsilon \cup \varepsilon_1} \ (\varepsilon\text{-}\textsc{Import3})$$

Figure 3.26: A third rule for type-and-effect checking `import` expressions.

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 $\{\texttt{File}\} \rightarrow \texttt{Unit}$, then we need $\{\texttt{File}.*\}$ in ε. To do this, we add a new premise $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon$. `hofx` 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: $\texttt{import}(\varnothing) \ x = \texttt{unit in} \ \lambda\texttt{f} : \texttt{File}.\ \texttt{f.write}$.

The unannotated code types via the judgement $\texttt{x} : \texttt{Unit} \vdash \lambda\texttt{f} : \texttt{File}.\ \texttt{f.write} : \{\texttt{File}\} \rightarrow \texttt{Unit}$. Its higher-order effects are $\texttt{ho-effects}(\texttt{annot}(\{\texttt{File}\} \rightarrow \texttt{Unit}, \varnothing)) = \{\texttt{File}.*\}$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$, so the example safely rejects.

The final version of ε-IMPORT is given in Figure 3.27.

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

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

Figure 3.27: 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 $\texttt{Unit} \rightarrow \texttt{Unit}$. ε-IMPORT would annotate `log` so it has the type $\texttt{Unit} \rightarrow_{\{\texttt{File}.*, \texttt{Socket}.*\}} \texttt{Unit}$. 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 11 (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`. Some lemmas are new, so we shall prove all of their 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 $\hat{\Gamma} \vdash x : \hat{\tau}\ \texttt{with}\ \varepsilon$ then $\varepsilon = \varnothing$.*
2. *If $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}\ \texttt{with}\ \varepsilon$ then $\varepsilon = \varnothing$.*
3. *If $\hat{\Gamma} \vdash \hat{v} : \{\bar{r}\}\ \texttt{with}\ \varepsilon$ then $\hat{v} = r$ and $\{\bar{r}\} = \{r\}$.*
4. *If $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2\ \texttt{with}\ \varepsilon$ 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, $[\hat{v}/y]$(import(ε_s) $y = \hat{e}_i$ in e) = import(ε_s) $y = [\hat{v}/x]\hat{e}_i$ in e. The result follows by applying the inductive assumption to $[\hat{v}/x]\hat{e}_i$. □

The progress theorem also has an extra case, and the proof is routine.

Theorem 12 (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 $\hat{e}$ = import(ε_s) $x = \hat{e}_i$ 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 $\hat{\tau}_i <:$ annot(erase($\hat{\tau}_i$), ε). This result is given as the pair of lemmas below.

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

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

Proof. By simultaneous induction on derivations of ho-safe($\hat{\tau}, \varepsilon$) and 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 import(ε_s) $x = \hat{v}_i$ in $e \longrightarrow [\hat{v}_i/x]$annot($e, \varepsilon_s$) $\mid \varnothing$. Since x : erase($\hat{\tau}$) $\vdash e : \tau$, it is reasonable to expect that (1) $\hat{\Gamma} \vdash$ annot(e, ε_s) : annot(τ, ε_s) with ε_s would be true, because although annot(e, ε_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 $\hat{\Gamma} \vdash [\hat{v}_i/x]$annot($e, \varepsilon_s$) : annot($\tau, \varepsilon_s$) with ε_s by the substitution lemma; that it does hold is the subject of the following lemma.

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

1. $\hat{\Gamma} \vdash \hat{v}_i : \hat{\tau}_i \texttt{ with } \varnothing$
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 $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s$*.*

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 this Γ because some effects *look* ambient in certain sub-scopes of e, and we need to keep track of them. For example, `f.write` exercises ambient authority over whatever resource is bound to f, but when enclosed by an appropriate abstraction like λ`f` : `File`. `f.write`, f is no longer ambient. Proving the lemma requires us to inductively step into the bodies of functions, at which point certain capabilities look ambient and need to be bound in the context — therefore, 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 $\Gamma = \varnothing$ then $\varepsilon' = \varepsilon_s$ and applying the lemma gives the judgement $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau_i, \varepsilon_s) \texttt{ with } \varepsilon_s$. This is ultimately the judgement we want, so when we apply the annotation lemma, we choose $\Gamma = \varnothing$.

The proof of the annotation lemma is quite long, but a sketch 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 $\Gamma, y : \texttt{erase}(\hat{\tau}_i), x : \tau_2 \vdash e_{body} : \tau_3 \texttt{ with } \varepsilon_s$. Apply the inductive hypothesis to e_{body} with $\Gamma, x : \tau_2$ as the context in which e_{body} typechecks, noting that the premises are satisfied because $\texttt{ho-effects}(\texttt{annot}(\tau,))\varnothing \subseteq \varepsilon_s$ implies $\texttt{fx}(\texttt{annot}(\tau_2, \varnothing) \subseteq \varepsilon_s$. 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 $\varepsilon_s \cup \{\bar{r}.\pi\}$. By considering where the binding is in $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}$, conclude that $\{\bar{r}.\pi\} \subseteq \varepsilon_s$.

□

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

Theorem 13 (CC Preservation). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}$ ε_A and $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$, then $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B$ $\texttt{with}$ 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 $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}$ ε_A, and then on $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$.

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

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

Theorem 14 (CC Single-step Soundness). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}$ ε_A and $\hat{e}_A$ is not a value, then $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$, where $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B$ $\texttt{with}$ ε_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 15 (CC Multi-step Soundness). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}$ ε_A and $\hat{e}_A \longrightarrow^* e_B \mid \varepsilon$, then $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B$ $\texttt{with}$ ε_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 in practice by presenting several 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 enable reasoning about the use of effects. The language is based on *Wyvern*, a pure, object-oriented, capability-safe language, with first-class modules.

In section 4.1. we discuss how the translation from Wyvern to CC will work, and what simplifying assumptions are made in our examples. This also serves as a gentle introduction to Wyvern's syntax. A variety of scenarios are then explored in section 4.2.

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 and forms 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*. We are going to introduce several rules to show a Wyvern program might be considered syntactic sugar for an CC program, and translate examples by desugaring according to our rules.

4.1.1 Unit

Unit is a type inhabited by exactly one value. It conveys the absence of information; in CC an operation call on a resource literal reduces to unit for this reason. We define unit $\overset{\text{def}}{=} \lambda x : \varnothing . x$. The unit literal is the same in both annotated and naked code. In annotated code, it has the type Unit $\overset{\text{def}}{=} \varnothing \rightarrow_{\varnothing} \varnothing$, while in naked code it has the type Unit $\overset{\text{def}}{=} \varnothing \rightarrow \varnothing$. While these are technically two seperate types, we will not distinguish between the annotated and naked versions, simply 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 $\texttt{let } x = \hat{e}_1 \texttt{ in } \hat{e}_2$ first binds the value $\hat{e}_1$ to the name x and then evaluates $\hat{e}_2$. We can generalise by allowing $\hat{e}_1$ to be a non-value, in which case it must first be reduced to a value. If $\Gamma \vdash \hat{e}_1 : \hat{\tau}_1$, then $\texttt{let } x = \hat{e}_1 \texttt{ in } \hat{e}_2 \overset{\text{def}}{=} (\lambda x : \hat{\tau}_1.\hat{e}_2)\hat{e}_1$. Note that 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.

`let` expressions can be used to sequence computations. Used in this way, the `let` expression simply names the results of the intemediate steps and then ignores them in its body. When we ignore the result of a computation we shall bind it to _ instead of a real name, to suggest the result isn't important and prevent the naming of unused variables.

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)

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

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

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

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

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

$$\frac{\hat{e}_1 \longrightarrow \hat{e}_1' \mid \varepsilon_1}{\texttt{let}\ x = \hat{e}_1\ \texttt{in}\ \hat{e}_2 \longrightarrow \texttt{let}\ x = \hat{e}_1'\ \texttt{in}\ \hat{e}_2 \mid \varepsilon_1}\ (\text{E-}\textsc{Let}1)$$

$$\frac{}{\texttt{let}\ x = \hat{v}\ \texttt{in}\ \hat{e} \longrightarrow [\hat{v}/x]\hat{e} \mid \varnothing}\ (\text{E-}\textsc{Let}2)$$

Figure 4.3: Derived `let` rules.

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 seperate file. Note how pure modules are declared with the `module` keyword, while resource modules are declared with the `resource module` keywords.

```
module PureMod

def tick(f: {File}):Unit
  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.

Wyvern is capability-safe, so resource modules must be instantiated with the capabilities they require. In 4.4, `ResourceMod` requests the use of a `File` capability, which must be supplied to it from someone already possessing it. Modules are behaving like objects in this way, because they require explicit instantiation. 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. They are then propagated by the top-level entry point.

```
require File
instantiate PureMod
instantiate ResourceMod(File)

def main():Unit
  PureMod.tick(File)
  ResourceMod.tick()
```

Figure 4.5: Definition of two modules, one pure and the other resourceful.

Before explaining our translation of Wyvern programs into `CC`, we must explain several simplifications made in all of our examples which enable our particular desugaring.

Objects are only ever used in the form of modules. Modules only ever contain functions and other modules, and have no mutable fields. The examples contain no recursion or self-reference, including a module invoking its own functions. Modules will not reference each other cyclically. Lastly, modules only contain one function definition. Despite these simplifications, the chosen examples will highlight the essential aspects of `CC`.

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 then it will take `Unit` as its argument. The constructor function for `M` is called `MakeM`. A function is then defined which represents the `main` function, which is the entry point into the program. This `main` function will instantiate all the modules by invoking the constructor functions, and then execute the body of code in main. Finally, the main function is invoked with the primitive capabilities it needs.

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 defiend on lines 11-15, which invokes the constructors and then runs the body of the `main` method. Lastly, line 17 starts everything off by invoking `Main` with the initial set of capabilities, which in this case is just `File`.

```
let MakePureMod =
  λx:Unit.
    λf:{File}. f.append
in

let MakeResourceMod =
  λf:{File}.
    λx:Unit. f.append
in

let MakeMain =
  λf:{File}.
    λx: Unit.
      let PureMod = (MakePureMod unit) in
      let ResourceMod = (MakeResourceMod f) in
      let _ = (PureMod f) in (ResourceMod unit) in

(MakeMain File) unit
```

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

Unannotated modules are modelled with `import`, where the unannotated body of the module is the unannotated body of the `import`. Most examples involving unannotated code will not typecheck because of a mismatch between the selected authority of `import` and the annotations on the client using unannotated code. Where an example involves unannotated code, the selected authority will be determined by what constraints are imposed by the client. For example, if the client only expects the `File.append` effect and executes some unlabelled code, the corresponding `import` expression will select `File.append`.

4.2 Examples

We now present several examples to show how the capability-based reasoning of `CC` can assist in reasoning about the effects of a program. We also hope to convince the reader that the rules of `CC` have practical worth, and could be used to enrich existing capability-safe languages in a straightforward and routine manner.

The format of each section is as follows. A program is introduced which exhibits some bad behaviour or demonstrates a particular story about software development. The language used is *Wyvern*; a pure, object-oriented, capability-safe language with first-class modules-as-objects. We show how the Wyvern program can be written as a corresponding CC program and sketch a derivation showing how the rules of CC and a sketch a derivation showing how the rules of CC would solve the relevant problem.

We take some shortcuts with the translation of Wyvern into CC. Our “objects” are really functions. The particular examples we have chosen only involve modules which export a single function and make no use of self-reference, so no important expressive properties are lost by treating Wyvern objects as functions.

4.2.1 API Violation

In the first example there is a single primitive capability called `File`, which is passed into the program when it begins execution, perhaps from the system environment or a virtual machine. There is a `Logger` module which possess this capability and exposes a single function `log` which incurs `File.write` when executed. The `Client` module possesses the `Logger` module as a capability. Its function `run` will invoke `Logger.log`, incurring both `File.write`; however, the client's annotation is expecting no 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` $\rightarrow_{\varnothing}$ `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`.

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

At line 12, when typing `LoggerModule`, an application of ε-App gives the judgement $\texttt{f} : \{\texttt{File}\} \vdash \texttt{LoggerModule} : \texttt{Unit} \rightarrow_{\texttt{File.write}} \texttt{Unit}\ \texttt{with}\ \varnothing$. At line 13 the same rule gives $\texttt{f} : \{\texttt{File}\} \vdash \texttt{ClientModule} : (\texttt{Unit} \rightarrow_{\varnothing} \texttt{Unit}) \rightarrow \texttt{Unit}\ \texttt{with}\ \varnothing$. Now at line 14, when attempting 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`, so 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? The answer is not immediately by inspecting the client's source-code, because it depends on what effects are incurred by `Logger.log`. A capability-based argument goes as follows: because the client code can typecheck needing only `Logger`, then the effects presented by `Logger` are an upper-bound on the effects of the client.

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 $\texttt{File.append}$ as its authority. This is exactly the effects of the logger, i.e. $\texttt{effects}(\texttt{Unit} \rightarrow_{\texttt{File.append}} \texttt{Unit}) = \{\texttt{File.append}\}$. 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.

In such a small example the client could simply inspect the source code of `Logger` to determine what effects it might have. Several situations can make this impossible or tedious. First, the manual approach loses efficiency when the system involves many modules of large size across code-ownership boundaries; capability-based reasoning tells you automatically. Second, the source code of `Logger` might be obfuscated or unavailable, and the only useful information is that given by its signature. Lastly, the client may not care about effects in this situation; the program may be a quick-and-dirty throwaway, in which case it is nice that the capability-based reasoning still accepts the client code without annotations.

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 $\varnothing$.

```
1 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 unlabelled 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 the `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

```

```
5  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 $\lambda\texttt{f} : \texttt{File}.\texttt{f}.\texttt{append}$, but encapsulated within an import expression selecting its authority as $\varnothing$ on line 5, to be consistent with the annotation on Client.run. Nothing is being imported, 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.

```
1  let MakeClient =
2    (λx: Unit.
3      let MakeLogger =
4        (λx: Unit.
5          import({File.⋆}) y=unit in
6            λf: {File}. f.append) 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
```

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 incur $\{\texttt{File}.*\}$ when invoked. This is true of the function, since it can do anything with the File it is given, but for the unannotated code this is a vast overapproximation. 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. This highlights a drawback in the type system: its approximations may include effects which cannot happen.

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.

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

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

```
1 resource module Main
2 require File
3 instantiate Plugin
4
5 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 type-check 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 ∅, 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 λ`f` : {`File`}. `LoggerModule f` are `File`.∗, which is not contained in the selected authority.

```
1 let MakePlugin =
2   (λx: Unit.
3     let MakeMalicious =
4       (λx: Unit.
5         import(∅) y=unit in
6           λf: {File}. f.append) in
7     let LoggerModule = MakeLogger unit in
8     λf: {File}. LoggerModule f) in
```

```

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

(MakeMain File) unit
```

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

```
module Malicious

def log(f: Unit → File):Unit
  f().read
```

```
module Plugin
instantiate Malicious

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

```
resource module Main
require File
instantiate Plugin

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 $\tau = (\texttt{Unit} \rightarrow \{\texttt{File}\}) \rightarrow \texttt{Unit}$. Then $\texttt{annot}(\tau, \varepsilon) = (\texttt{Unit} \rightarrow_{\varnothing} \{\texttt{File}\}) \rightarrow_{\varnothing} \texttt{Unit}$ and $\texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon)) = \{\texttt{File}.*\}$. Thus, the premise cannot be satisfied and the example safely rejects.

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

Chapter 5

Evaluation

5.1 Related Work

Fengyun Liu has explored how a capability-safety can be used to determine which sections of code are pure [10]. His approach develops a lambda calculus with two type-constructors for building free and stoic functions. Free functions may ambiently capture capabilities, but stoic functions may not. For a stoic function to have effects, it must be explicitly given the capability for that effect. These functions are small, capability-safe pockets that enable the type system to determine purity. 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.

By contrast, our work is motivated by how capabilities are propagated and exercised, and how language-design features might inform sofwate design. Unlike Liu's System F-Impure, `CC` has no effect-polymorphism. However, our work has more fine-grained detail about those effects incurred by a particular function, and distinguishes higher-order effects from the non-higher-order kind.

5.2 Future Work

A limitation to practical adoption of `CC` is that it is not Turing complete — it has no general recursion, nor recursive types. Extending `CC` to include these features would bring it up to par with real programming languages. Extending `CC` to accommodate these features is expected to be routine, but has not been done.

We have seen that approximating some unannotated code by every effect it exercises can lead to very conservative overapproximations. Section 4.2.4. illustrates with $\texttt{import}(\texttt{File}.*)\ x = \mathit{unit}\ \texttt{in}\ \lambda f : \{\texttt{File}\}.\ f.\mathit{write}$ which is a piece of effect-less, unannotated code that returns an effectful function. However, the effectless code is approximated as incurring $\texttt{File}.*$. By a careful analysis of which effects are sourced directly or in a higher-order fashion,t he rules of `CC` might be amended to give a better approximation.

The current theory contains no notion of polymorphic effects. As an example, consider $\lambda x : \texttt{Unit} \rightarrow_{\varepsilon} \texttt{Unit}.\ x\ \texttt{unit}$, where ε is free. Invoking this particular function would incur every effect in ε. In CC, ε is only allowed to be a concrete set of effects. It has no way to define functions which are parametrised by effect-sets. Developing an extension which can handle polymorphic effects would be a valuable contribution, and improve the stock of CC as a practical and expressive effect system.

5.3 Conclusion

CC is a lambda calculus with a notion of primitive capabilities (resources) and the operations on them. It contains an annotated sublanguage and an import construct which allows developers to nest unannotated code inside annotated code. The typing rule for import is defined according to capability-safe principles, to prohibit the exercise of ambient authority. The result is a sound type-and-effect system which can safely approximate the effects of an unannotated body of code by inspecting what capabilities are passed into it. Section 4 has shown how CC can express practical examples.

There are some limitations to CC, such as its limited expressiveness, overapproximation when unannotated code is returning a function, and lack of polymorphic effects. These are all interesting avenues of future work that would enrich CC and our collective understanding of the relation between effects and capabilities.

Appendix A

OC Proofs

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

1. *If* $\Gamma \vdash x : \tau \texttt{ with } \varepsilon$ *then* $\varepsilon = \varnothing$.
2. *If* $\Gamma \vdash v : \tau \texttt{ with } \varepsilon$ *then* $\varepsilon = \varnothing$.
3. *If* $\Gamma \vdash v : \{\bar{r}\} \texttt{ with } \varepsilon$ *then* $v = r$ *and* $\{\bar{r}\} = \{r\}$.
4. *If* $\Gamma \vdash v : \tau_1 \rightarrow_{\varepsilon'} \tau_2 \texttt{ with } \varepsilon$ *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 $\tau_1 \rightarrow_{\varepsilon'} \tau_2$ is ε-ABS. Its premises imply the result.

□

Theorem 16 (OC Progress). *If* $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$ *and* e *is not a value or variable, then* $e \longrightarrow e' \mid \varepsilon$*, for some* e', ε.

Proof. By induction on $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$.

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, $\Gamma \vdash v_1 : \tau_2 \rightarrow_{\varepsilon'} \tau_3 \texttt{ with } \varnothing$, 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 $\Gamma, x : \tau' \vdash e_1 : \tau_2 \rightarrow_{\varepsilon_3} \tau_3$ `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, $\Gamma \vdash [v/x]e_1 : \tau_2 \rightarrow_{\varepsilon_3} \tau_3$ `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 17 (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 18 (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 19 (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, $\Gamma \vdash e_C : \tau_C \texttt{ with } \varepsilon_C$ and $\Gamma \vdash e_B : \tau_B \texttt{ with } \varepsilon_B$, 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 20 (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 $\hat{e} = \texttt{import}(\varepsilon_s)\ x = \hat{e}_i$ in e. If $\hat{e}_i$ is a non-value then $\hat{e}_i \longrightarrow \hat{e}'_i \mid \varepsilon$ by inductive assumption and $\texttt{import}(\varepsilon_s)\ x = \hat{e}_i$ in $e \longrightarrow \texttt{import}(\varepsilon_s)\ x = \hat{e}'_i$ in $e \mid \varepsilon$ by E-MODULE1. Otherwise $\hat{e}_i = \hat{v}_i$ is a value and $\texttt{import}(\varepsilon_s)\ x = \hat{v}_i$ 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. $\hat{e} = \texttt{import}(\varepsilon_s)\ x = \hat{e}_i$ 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. $\hat{\Gamma}, y : \hat{\tau}' \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}_i$ 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) $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_i : \hat{\tau}_i \texttt{ with } \varepsilon_i$. Then by ε-MODULE $\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ y = [\hat{v}/x]\hat{e}_i \texttt{ in } e : \texttt{annot}(\tau_i, \varepsilon_s) \texttt{ with } \varepsilon_s \cup \varepsilon_i$. 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: $\hat{\tau} = \hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2$, $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ 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 2** to $\hat{\tau}_2$ and **lemma 3** 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 2** 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 3** to $\hat{\tau}_1$.

Case: $\hat{\tau} = \hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2$, $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{safe}(\hat{\tau}, \varepsilon)$ 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 3** to $\hat{\tau}_2$ and **lemma 2** 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 3** 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 2** to $\hat{\tau}_1$.

□

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

1. $\hat{\Gamma} \vdash \hat{v}_i : \hat{\tau}_i \; \texttt{with} \; \varnothing$
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 $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s) \; \texttt{with} \; \varepsilon_s$.

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, $y : \hat{\tau}_i \vdash x : \hat{\tau}_i \; \texttt{with} \; \varnothing$. By definition $\texttt{annot}(x, \varepsilon_s) = x$, so (5) $y : \hat{\tau}_i \vdash \texttt{annot}(x, \varepsilon_s) : \hat{\tau}_i \; \texttt{with} \; \varnothing$. 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 $y : \hat{\tau}_i \vdash \texttt{annot}(x, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s) \; \texttt{with} \; \varepsilon_s$. 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, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \lambda x : \texttt{annot}(\hat{\tau}_2, \varepsilon_s).\texttt{annot}(e_{body}, \varepsilon_s) : \texttt{annot}(\hat{\tau}_2, \varepsilon_s) \rightarrow_{\varepsilon_s} \texttt{annot}(\hat{\tau}_3, \varepsilon_s)\ \texttt{with}\ \varnothing$. 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, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}_i \vdash \texttt{annot}(e_1, \varepsilon_2) : \texttt{annot}(\tau_2, \varepsilon_s) \rightarrow_{\varepsilon_s} \texttt{annot}(\tau_3, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$ 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 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 subderivation $\Gamma, y : \texttt{erase}(\hat{\tau}_i) \vdash e_1 : \{\bar{r}\}$. 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 21 (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 22 (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 23 (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] 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).

[3] 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).

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

[5] 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).

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

[7] KLEENE, S. Recursive predicates and quantifiers. *Journal of Symbolic Logic 8*, 1 (1943), 32–34.

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

[9] KURILOVA, D., POTANIN, A., AND ALDRICH, J. Modules in wyvern: Advanced control over security and privacy. In *Symposium and Bootcamp on the Science of Security* (2016). Poster.

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

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

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

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

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

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

[22] 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.

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

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

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

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