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

Privilege separation and least authority are principles for developing safe software, but existing languages offer insufficient techniques for allowing developers and architects to make informed design choices enforcing them. Languages adhering to the object-capability model impose constraints on the ways in which privileges are used and exchanged, giving rise to a form of lightweight effect-system. This effect-system allows architects and developers to make more informed choices about whether code from untrusted sources should be used. This paper develops an extension of the simply-typed lambda calculus to illustrate the ideas and proves it 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 **REFERENCE**. 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, 11, 24, 20]. 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, 23]. 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 [18]. For ex-

ample, 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 [19]. 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 [14]. 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" [13]. 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 [12].

Effect systems were introduced by Lucassen and Gifford for the purposes of optimising pure code [**?**]. They have also been applied to problems such as determining which functions might be invoked in a program [22], or determining which regions in memory may be accessed or updated [21]. 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 [17]. 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 defini-

tion 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 coevr, 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 describ-

ing 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, $3 + 2$, `false` $\vee$ `true`, $3 \vee$ `false`, `true` $+$ `false`, `let` $x = 3$ `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.

e	$::=$		$exprs:$
	$\mid$	x	$variable$
	$\mid$	$e + e$	$addition$
	$\mid$	$e \vee e$	$disjunction$
	$\mid$	`let` $x = e$ `in` e	$let\ expr.$
v	$::=$		$values:$
	$\mid$	l	`Nat` $constant$
	$\mid$	b	`Bool` $constant$

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 `let` $x = 1$ `in` (`let` $x = 2$ `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 `let` $x = 1$ `in` (`let` $x = 2$ `in` $x + z$). In some sense, this is an equivalent program to `let` $x = 1$ `in` (`let` $y = 2$ `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 [16, 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 `let` $x = 1 + 1$ `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, `false`$\vee 3 \longrightarrow 3$ by E-OR3, which is nonsense. $(1+1)+$`false` $\longrightarrow 2+$`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 [16, p. 39].

stuck because $+$ is an operation on numbers, and `false` is a boolean. Another example is `x` $+ 1$, which gets stuck because x is undefined.

We often want to consider those programs which satisfy certain well-behavedness properties. One such property is that of being *well-typed*: if a program is well-typed then during execution it will never get *stuck* due to type-errors. 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 `let` $x = 1$ `in` $x + 1$, where in typing $x + 1$, we need to know the type of x.

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

Figure 2.5: Grammar for the type system of EBL.

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

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

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

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

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

Figure 2.6: Inference rules for typing arithmetic expressions.

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 dis-

junction 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.

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

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

There are some pesky technicalities about typing contexts which need to be addressed. Though we have defined Γ syntactically as a sequence of variable-type pairs, we really want to treat it 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.

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

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

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.

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

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

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.

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

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

This report is only 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" [16, 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. In this section we present a variation of $\lambda^{\rightarrow}$ with subtyping and summarise its basic properties. Various

λ-calculi serve as the basis for numerous functional programming languages, including `CC`. This section gives us an opportunity to familiarise ourself with $\lambda^{\to}$ to help introduce `CC`.

$$
\begin{array}{rcll}
e & ::= & & exprs: \\
 & | & x & variable \\
 & | & e\ e & application \\
 & | & v & value
\end{array}
\qquad
\begin{array}{rcll}
v & ::= & & values: \\
 & | & \lambda x:\tau.e & abstraction
\end{array}
$$

$$
\begin{array}{rcll}
\tau & ::= & & types: \\
 & | & B & base\ type \\
 & | & \tau \to \tau & arrow\ type
\end{array}
\qquad
\begin{array}{rcll}
\Gamma & ::= & & contexts: \\
 & | & \varnothing & empty\ ctx. \\
 & | & \Gamma, x:\tau & var.\ binding
\end{array}
$$

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

Types in $\lambda^{\to}$ are either drawn from a set of base types B, or constructed using $\to$ ("arrow"). Given types τ_1 and τ_2, $\to$ can be used to compose a new type, $\tau_1 \to \tau_2$, which is the type of function taking τ_1-typed terms as input to produce τ_2-typed terms as output. For example, given $B = \{\texttt{Bool}, \texttt{Int}\}$, the following are examples of valid types: `Bool`, `Int`, `Bool` $\to$ `Bool`, `Bool` $\to$ `Int`, `Bool` $\to$ (`Bool` $\to$ `Int`). Arrow is right-associative, so `Bool` $\to$ `Bool` $\to$ `Int` = `Bool` $\to$ (`Bool` $\to$ `Int`). "Arrow-type" and "function-type" will be used interchangeably.

In addition to variables, there are function definitions ("abstraction") and the application of a function to an expression ("application"). For example, $\lambda x : \texttt{Int}.x$ is the identity function on integers. $(\lambda x : \texttt{Int}.x)3$ is the application of the identity function to the integer literal 3. $(\lambda x : \texttt{Int}.x)\texttt{true}$ is the applciation of the identity function to a boolean literal, which is syntactically valid, but as we'll see is not well-typed. A more drastic example is `true` 3, which is trying to apply `true` to 3. Again, this is a syntactically valid term, but not well-typed because `true` is not a function.

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

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

$$\frac{\Gamma \vdash e_1:\tau_2 \to \tau_3 \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash e_1\ e_2:\tau_3}\ \text{(T-APP)} \qquad \frac{\Gamma \vdash e:\tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e:\tau_2}\ \text{T-SUBSUME}$$

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

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

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

Static rules for $\lambda^{\rightarrow}$ are summarised in Figure 2.8. 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. T-APP states that an application is well-typed if the left-hand expression is a function (has an arrow-type $\tau_2 \rightarrow \tau_3$) and the right-hand expression has the same type as the function's input (τ_2).

T-SUBSUME is the rule which says you may a type a term more generally as any of its supertypes. For example, if we had base types `Int` and `Real`, and a rule specifying `Int` <: `Real`, a term of type `Int` can also be typed as `Real`. This allows programs such as $(\lambda x : \texttt{Real}.x)\ 3$ to type, as shown in Figure 2.13.

$$\dfrac{\dfrac{\dfrac{}{x : \texttt{Real} \vdash x : \texttt{Real}}\ \text{(T-VAR)}}{\vdash \lambda x : \texttt{Real}.x : \texttt{Real} \rightarrow \texttt{Real}}\ \text{(T-ABS)} \qquad \dfrac{\vdash 3 : \texttt{Int} \quad \texttt{Int} <: \texttt{Real}}{\vdash 3 : \texttt{Real}}\ \text{(T-SUBSUME)}}{\vdash (\lambda x : \texttt{Real}.x)\ 3 : \texttt{Real}}\ \text{(T-APP)}$$

Figure 2.13: Derivation tree showing how T-SUBSUME can be used.

The only subtyping rule we provide is S-ARROW, which describes when one function is a subtype of another. Note how the subtyping relation on the input types is reversed from the subtyping relation on the functions. This is called *contravariance*. Contrast this with the relation on the output type, which preserves the order. That is called *covariance*. Arrow-types are contravariant in their input and covariant in their output.

This presentation has no subtyping rules without premises (axioms), which means there is no way to actually prove a particular subtyping judgement. In practice, we add subtyping axioms for the base-types we have chosen as primitive in our calculus. For example, given base types `Int` and `Real`, we might add `Real` <: `Int` as a rule. This is largely an implementation detail particular to your chosen set of base-types, so we give no subtyping axioms here (but will later when describing `CC`).

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

$[v/y]x = v$, if $x = y$
$[v/y]x = x$, if $x \neq y$
$[v/y](\lambda x : \tau.e) = \lambda x : \tau.[v/y]e$, if $y \neq x$ and y does not occur free in e
$[v/y](e_1\ e_2) = ([v/y]e_1)([v/y]e_2)$

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

Substitution in $\lambda^{\rightarrow}$ follows the same conventions as it does in `EBL`. Substitution on an application is the same as substitution on its sub-expressions. Substitution on a function involves substitution on the function body.

Applications are the only reducible expressions in $\lambda^{\rightarrow}$. Such an expression is reduced by first reducing the left subexpression (E-APP1). For a well-typed expression, this will

$\boxed{e \longrightarrow e}$

$$\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1' e_2 \mid \varepsilon}\ \text{(E-App1)} \qquad \frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1 e_2 \longrightarrow v_1 e_2' \mid \varepsilon}\ \text{(E-App2)}$$

$$\frac{}{(\lambda x : \tau . e) v_2 \longrightarrow [v_2/x]e \mid \varnothing}\ \text{(E-App3)}$$

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

always be a function. Once that is a value, the right subexpression is reduced (E-App2). When both subexpressions are values, the right subexpression replaces the formal argument of the function via substitution. The multi-step rules for $\lambda^{\rightarrow}$ are identical to those in `EBL`.

The soundness property for $\lambda^{\rightarrow}$ is as follows.

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* $\tau_B <: \tau_A$*.*

Note how with the inclusion of subtyping rules, the type after reduction can get more specific than the type before reduction, but never less specific. This is in contrast to `EBL`, where the type remains the same.

$\lambda^{\rightarrow}$ is also strongly-normalizing, meaning that well-typed terms always halt i.e. they eventually yield a value. As a consequence it is *not* Turing complete, meaning there are certain computer programs which cannot be written in $\lambda^{\rightarrow}$. By comparison, the *untyped* λ-calculus is known to be Turing complete [7]. The essential ingredient missing from $\lambda^{\rightarrow}$ is a means of general recursion. In mainstream languages such as Java, this is realised by constructs like the `while` loop; in the untyped λ-calculus by the Y-combinator. $\lambda^{\rightarrow}$ can be made Turing-complete by adding a `fix` operator which mimics the Y-combinator.

Turing-completeness is an essential property for practical, general-purpose programming languages. However, the key contribution of this report is in the static rules of `CC`, and not the expressive power of its dynamic semantics. Therefore we acknowledge this practical short-coming, but leave the basis of `CC` as a Turing-incomplete language to reduce the number of rules and simplify its presentation.

Revisit this depending on how you encode types and stuff in `CC`

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 [14]. To illustrate, we summarise the static rules of SEA (Side-Effect Analysis), which is a lambda calculus for reasoning about the set of memory cells that are written or read during execution [14]. Our summary is simplified for presentation purposes.

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 2.16. 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.

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

Figure 2.16: 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 2.17.

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

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

Figure 2.17: Grammar for effects and regions of SEA.

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. $\texttt{ref}(\tau, \rho)$ 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 2.18.

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

Figure 2.18: Grammar for SEA types.

There is a single judgement in SEA, which has the form $\Gamma \vdash e : \tau$ with Φ. 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 2.21.

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 the expressino being assigned, Φ. T-NEW is well-typed if the initial expression e_1 of x is

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

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

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

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

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

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

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

Figure 2.19: Static rules for SEA.

well-typed, and the same environment with a new binding $x : \texttt{ref}(\tau_1, \pi)$ can type the rest of the code e_2. The effects incurred by the new expression are those incurred by reducing the initial expresion (Φ_1) and those incurred by reducing the rest of the code (Φ_2).

The rules of SEA now give us the ability to determine which locations in memory are instantiated, modified, or accessed — and we do not have to execute the program to find out! As an example, consider the program $e = \texttt{new}_{l_1} x = 3 \texttt{ in } x := 5$, which initialises a reference at location l_1 with 3, and then updates it to 5 . This can be typed as $\vdash e : \texttt{Nat with } \{l_1 :=\}$; a derivation tree is given in Figure 2.20.

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

Figure 2.20: Derivation tree for $\texttt{new}_{l_1} x = 3 \texttt{ in } x := 5$.

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 $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$ 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, and the effects incurred along both branches. Only branch is executed during runtime,

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

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

Figure 2.21: Static rules for SEA.

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]. A piece of code S has *authority* over a capability C if it can directly invoke the operations granted by C; it has *transitive authority* if it can indirectly invoke the operations endowed by a capability C (for example, by deferring to another piece of code with authority over C). In the capability model, authority can only proliferate in the following ways [13]:

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". The initial set of capabilites are passed into the program at the beginning of execution by the system environment or virtual machine, and grant operations over *resources* in the system environment. A capability is either one of these initial capabilities, or a function or object which captures (potentially transitive) authority over an existing capability. For example, an initial capability `File` might grant read and write access to a particular resource in the system environment. Usually we conflate initial capabilities and they system resources they grant access to, referring to both as resources. Another capability might be a `Logger`, which presents a confined subset of operations on `File`, such as allowing its bearer to perform `append` operations, but not `read` or `write`.

The proliferation rules impose constraints on how capabilities may spread throughout a program. The result is that any component of a program which needs to use a

particular system resource must either possess a capability for it, or be given a capability as a function argument, so gaining authority is an explicit process. In a language not adhering to the capability model, the implicit exercise of authority is known as *ambient authority*. Figure 2.22 demonstrates an example of ambient authority 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.

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

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

```
import java.util.List;

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

Figure 2.22: `Main` exercises ambient authority over a `File` capability.

Another way to exercise ambient authority is through global state: if a capability is stored inside a global variable then any component is able to access and use its operations without having been given the capability. Therefore, languages adhering to the capability model must disallow global state and unrestricted imports.

Ambient authority is also a challenge to POLA 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 which often surround untrusted code and code ownership.

Languages adhering to the capability model often have first-class modules, meaning objects and modules are treated in a uniform manner. This means modules, like objects, must be instantiated with those capabilites they require. They are also bound by the same proliferation rules constraining objects, which allosw the constraints of the capability model to be preserved across module boundaries. Java is an example of a mainstream language whose modules are not first-class. Scala has first-class modules [15], but is not capability-safe. Smalltalk is a dynamically-typed capability-safe language with first-class modules [2]. Wyvern is a statically-typed capability-safe language with first-class modules [9].

A language is *capability-safe* if it satisfies this capability model and disallows ambient authority. Some examples include E, Js, and Wyvern.

Chapter 3

Effect Calculi

In this section we introduce a pair of langauges: 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. They are type-sound, but also *effect-sound*, which is a new notion we introduce.

We then extend OC to obtain CC, which allows the nesting of unannotated code inside annotated code using a new `import` construct. An effect-sound inference can be made about what effects the unannotated code might incur by inspecting those capabilities passed into the unannotated code.

The high-level motivating examples in this section written in a *Wyvern*-like language. Wyvern is a capability-safe, pure, object-oriented language with first-class modules. 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 is an extension of $\lambda^{\rightarrow}$ with primitive capabilities and operations which can be invoked on them. A primitive capability encapsulates some system resource. For simplicity, we conflate the two and use the term resource to refer to primitive capabilities. An effect is a particular operation invoked on some resource. Every function-type 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.

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 3.1.

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

Figure 3.1: 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 simply exist throughout the duration of the program. They resource those initial capabilities passed into the program which endow operations upon some system resources. For example, a `File` or `Socket` might provide read-and-write operations on a particular file or socket in the system environment.

An operation call $e.\pi$ represents some primitive operation 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 3.2. 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.

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

Figure 3.2: 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; for example, `File.write`(“`msgToWrite`”). We make the assumption that all operations are null-ary and return a value. Furthermore, all operations are assumed to be valid on all resources.

An updated definition of `substitution` is given in Figure 3.3. The new cases are straightforward, but we 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 repalced with

an arbitrary expression, then 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`

$[v/y]x = v$, if $x = y$
$[v/y]x = x$, if $x \neq y$
$[v/y](\lambda x : \tau.e) = \lambda x : \tau.[v/y]e$, if $y \neq x$ and y does not occur free in e
$[v/y](e_1\ e_2) = ([v/y]e_1)([v/y]e_2)$
$[v/y](e_1.\pi) = ([v/y]e_1).\pi$

Figure 3.3: Substitution function in OC.

3.1.2 OC Dynamic Rules

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. The judgements for single-step reductions are summarised in Figure 3.4.

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

$$\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1'\ e_2 \mid \varepsilon}\ \text{(E-APP1)} \qquad \frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1\ e_2 \longrightarrow v_1\ e_2' \mid \varepsilon}\ \text{(E-APP2)} \qquad \frac{}{(\lambda x : \tau.e)v_2 \longrightarrow [v_2/x]e \mid \varnothing}\ \text{(E-APP3)}$$

$$\frac{e \to e' \mid \varepsilon}{e.\pi \longrightarrow e'.\pi \mid \varepsilon}\ \text{(E-OPERCALL1)} \qquad \frac{}{r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}}\ \text{(E-OPERCALL2)}$$

Figure 3.4: Single-step reductions in OC.

E-APP1 and E-APP2 incur whatever is the effect of reducing their subexpressions. E-APP3 incurs effects when it substitutes the actual value of the argument for its formal name in the function body. The first new single-step rule is E-OPERCALL1, which reduces the receiver of an operation call. The effects incurred are the effects incurred by reducing the receiver. The other new rule is E-OPERCALL2, which is reducing an operation call on a resource literal; then $r.\pi$ incurs the effect-set $\{r.\pi\}$.

From the judgements for single-step reduction we define judgements for multi-step reductions in Figure 3.5. 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.

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

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

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

Figure 3.5: Multi-step reductions in OC.

3.1.3 OC Static Rules

A grammar for the types of OC is given in Figure 3.6. Typing contexts are the same as in $\lambda^{\to}$. 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). 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.

τ	::=	*types* :	Γ	::=	*type ctx* :
	$\{\bar{r}\}$			$\varnothing$	
	$\tau \to_\varepsilon \tau$			$\Gamma, x : \tau$	

Figure 3.6: 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 `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 $\Gamma \vdash e : \tau$ `with` ε, 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 3.8.

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

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

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

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

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

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

Figure 3.7: Type-with-effect judgements in OC.

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 $\Gamma \vdash e : \{\texttt{File}, \texttt{Socket}\}$ `with` $\varnothing$. Then e could evaluate to `File`, in which case the actual runtime effect is `File`.π, or it could evaluate to `Socket`, in which case the actual runtime effect is `Socket`.π. 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 $\vdash$ `unit` : `Unit with` $\varnothing$ by a derived rule ε-UNIT. Definitions for this are given in Chapter 4.

The last rule, ε-SUBSUME, only makes sense in the presence of subtyping 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 3.8.

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

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

Figure 3.8: Subtyping judgements of OC.

The first subtyping rule is S-ARROW, which is similar to the rule for subtyping functions in $\lambda^{\to}$. The only addition is that the effects of the subtype must be contained in

the effects of the supertype. This is because any possible effect which might be incurred by the subtype should be expected by the supertype, otherwise you could supply an instance of the subtype and incur an effect the supertype's interface was not 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 $e = (\lambda \texttt{f} : \{\texttt{File}, \texttt{Socket}\}.\texttt{f.write})\ \texttt{File}$. The judgement $\vdash e : \texttt{Unit with } \{\texttt{File.write}, \texttt{Socket.write}\}$ 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 3.9. 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.

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

Figure 3.9: Derivation tree for $(\lambda \texttt{f} : \{\texttt{File}, \texttt{Socket}\}.\ \texttt{f.write})\ \texttt{File}$.

These rules can be used to determine if a piece of code is safe. For example, a function which uses a logger might be $\texttt{e} = \lambda \texttt{l} : \texttt{File} \to_{\texttt{File.append}} \texttt{Unit}.\ \texttt{l unit}$. Applying the rules to the logger implementation `l` gives an approximation to the effects it might incur. With 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 $(\texttt{get}\ \texttt{File}).\texttt{write}$. In the context $\Gamma = \texttt{File} : \{\texttt{File}\}$, the rule ε-APP can be used to approximate the effects of $(\texttt{f}\ \texttt{File}).\texttt{write}$ as $\{\texttt{File.write}, \texttt{Socket.write}\}$. By E-APP3 we have the reduction $(\texttt{get}\ \texttt{File}).\texttt{write} \longrightarrow \texttt{File.write} \mid \varnothing$. 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* $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ *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* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$*, for some* $e, \varepsilon, \tau_B, \varepsilon_B$*.*

Our approach to proving soundness wil be to show progress and preservation. These in turn rely on canonical forms and the substitution lemma, modified for OC. The new version of canonical forms states that resource-typed values are resource literals, and any typing judgement of a value will approximate the runtime effects as $\varnothing$. This result is not true if the rule used is ε-SUBSUME, so the lemma statement excludes judgements which use that rule. Progress follows from Canonical Forms.

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

- *If* $\Gamma \vdash v : \tau \texttt{ with } \varepsilon$ *then* $\varepsilon = \varnothing$*.*
- *If* $\Gamma \vdash v : \{\bar{r}\}$ *then* $v = r$ *for some* $r \in R$ *and* $\{\bar{r}\} = \{r\}$*.*

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

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

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

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

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

With this lemma, we can prove the preservation theorem.

Theorem 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 $e_1 : \tau_2 \to_\varepsilon \tau_3$ `with` ε_1 and $\Gamma \vdash e_2 : \tau_2$ `with` ε_2. If the reduction rule used was E-APP1 or E-APP2, then the result follows by applying the inductive hypothesis to e_1 and e_2 respectively. Otherwise the rule used was E-APP3. Then $(\lambda x : \tau_2.e)v_2 \longrightarrow [v_2/x]e \mid \varnothing$. By inversion on the typing rule for $\lambda x : \tau_2.e$ we know $\Gamma, x : \tau_2 \vdash e : \tau_3$ `with` ε_3. By canonical forms, $\varepsilon_2 = \varnothing$ because $e_2 = v_2$ is a value. Then by the substitution lemma, $\Gamma \vdash [v_2/x]e : \tau_3$ `with` ε_3. By canonical forms, $\varepsilon_1 = \varepsilon_2 = \varnothing = \varepsilon_C$. Therefore $\varepsilon_A = \varepsilon_3 = \varepsilon_B \cup \varepsilon_C$.

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

□

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

Theorem 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: Epsilon Calculus

OC requires every function to have its input type annotated — if we relax this requirement, can our type system say anything useful about pieces of unannotated code? In this section we introduce CC, which leverages capability-safe design to enable our type-system to approximate what effects might be incurred by unannotated code.

There are practical reasons to permit unannotated code in an effect-conscious language. Previous effect systems have been criticised for their verbosity, **citation needed** which might disincline a programmer from bothering to use them. Allowing the structured mixture of annotated and unannotated also permits one to rapidly prototype software in the unannotated sublanguage and then incrementally add effect annotations as they are needed, giving a trade-off between convenience and safety.

In general, reasoning about unannotated code is difficult because there are no constraints on what effects they might incur. Figure 3.10 demonstrates the issue with a Wyvern-like snippet of code. `someMethod` takes a function as input and executes it; but the effects of `f` depend on what particular implementation is passed to `someMethod`. Without more information, whether by means of a more complex type-system or more static annotations, there is no general way to know what effects might be incurred by `someMethod`.

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

Figure 3.10: What effects might be incurred by `someMethod`?

Therefore, CC permits the nesting of unannotated code when it is nested inside annotated code with the `import` expression. The typing rule for `import` expression, ε-IMPORT,

imposes a set of restrictions on what authority can be exercised by the unannotated code. This enables the type system to safely approximate the effects of unannotated code as those effects captured by the capabilities selected by enclosing `import` expressions. This is the key result of `CC`, which we formalise and prove in this section.

3.2.1 `CC` Grammar

The grammar of `CC` is essentially split into rules for annotated code and analogous rules for unannotated code. To distinguish the two, annotated types, expressions, and contexts always have a hat above them: $\hat{e}$, $\hat{\tau}$, and $\hat{\Gamma}$ instead of e, τ, and Γ.

The unannotated portion consists of programs made from the same building blocks as `OC`, but with the regular type-constructor $\rightarrow$ from $\lambda^{\rightarrow}$. Unannotated programs have no labels on their functions at all. They are also *deeply* unannotated: if a term is unannotated at the top-level, then every sub-term is also unannotated, so you cannot nest annotated code inside unannotated code. The corresponding grammar of types contains unannotated types τ, which are built from resource-stes and $\rightarrow$, and unannotated contexts Γ. An unannotated context can only map variables to unannotated types.

Except for the new `import` expression, the analogous rules for annotated programs and their surrounding meta-theory is the same as `OC`. Annotated types $\hat{\tau}$ are those built using the type constructors $\rightarrow_{\varepsilon}$, where ε is a concrete set of effects. The category of annotated contexts is $\hat{\Gamma}$, which binds variables to annotated types.

The interesting new form is `import`, which belongs to the annotated sublanguage. `import` introduces a name x with annotated definition $\hat{e}$ into a body of unannotated code e. ε is the set of effects selected by the unannotated code, so any resources and operation calls used in e must be declared in ε. `import` is the only means of nesting unannotated subterms inside annotated terms.

3.2.2 `CC` Dynamic Rules

Different approaches can be taken to defining the execution of an `CC`. One way is to define reductions for both annotated and unannotated programs, but this results in a lot of uninteresting rules which clutter the formalism. Another way is to define reductions for either annotated or unannotated programs, and translate programs into the appropriate form before executing them. We choose this approach, but the transformation happens during execution of the program, rather than before the program is executed. The idea is that whenever a piece of unlabelled code is encountered, the `import` expression surrounding it will have been evaluated to the point where we know what effects ε are being selected by the unannotated body e. At this point, we can annotate e with ε and continue executing this result.

To this end we define `annot` in Figure 3.12. This function takes a piece of unlabelled

$$
\begin{array}{lcll}
e & ::= & & exprs: \\
 & | & x & variable \\
 & | & v & value \\
 & | & e\ e & application \\
 & | & e.\pi & operation \\
 & & & \\
v & ::= & & values: \\
 & | & r & resource\ literal \\
 & | & \lambda x : \tau.e & abstraction \\
 & & & \\
\hat{e} & ::= & & labelled\ exprs: \\
 & | & x & \\
 & | & \hat{v} & \\
 & | & \hat{e}\ \hat{e} & \\
 & | & \hat{e}.\pi & \\
 & | & \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e & import \\
 & & & \\
\hat{v} & ::= & & labelled\ values: \\
 & | & r & \\
 & | & \lambda x : \hat{\tau}.\hat{e} &
\end{array}
\qquad
\begin{array}{lcll}
 & & & \\
\varepsilon & ::= & & effects: \\
 & | & \{\overline{r.\pi}\} & \\
 & & & \\
\tau & ::= & & types: \\
 & | & \{\bar{r}\} & \\
 & | & \tau \to \tau & \\
 & & & \\
\hat{\tau} & ::= & & labelled\ types: \\
 & | & \{\bar{r}\} & \\
 & | & \hat{\tau} \to_{\varepsilon} \hat{\tau} & \\
 & & & \\
\Gamma & ::= & & type\ ctx: \\
 & | & \varnothing & \\
 & | & \Gamma, x : \tau & \\
 & & & \\
\hat{\Gamma} & ::= & & labelled\ type\ ctx: \\
 & | & \varnothing & \\
 & | & \hat{\Gamma}, x : \hat{\tau} &
\end{array}
$$

Figure 3.11: Effect calculus.

code e and a set of effects ε and produces $\hat{e}$, obtained by labelling every arrow-type with ε. A version of this function is given for expressions, types, and contexts. Its dual is `erase`, which deletes all annotations from code. We will need `erase` in the definition of ε-IMPORT, so we give its definition here. Like `annot`, there are versions which delete the annotations from expressions, contexts, and types. However, `erase` is partial because it is not defined on `import` expressions.

It is worth mentioning that `annot` and `erase` operate on a purely syntactic level. Their definitions state no meaningful correspondence between the types and effects of a program before and after it has been annotated or erased. It remains for us to prove there is a meaningful correspondence in the particular way these functions are used in the rules for `import` expressions.

Finally, before giving the dynamic rules we must define a version of `substitution` for `CC`. As our dynamic rules are defined on annotated expressions, so too will `substitution` be defined. The definition, given in 3.13, is otherwise straight-forward. It has been updated for the new annotated notation and the new `import` expression. It retains the same

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

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

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

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

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

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

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

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

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

$$\begin{aligned}
&\texttt{erase}(r) = r \\
&\texttt{erase}(\lambda x : \hat{\tau}_1.\hat{e}) = \lambda x : \texttt{erase}(\hat{\tau}_1).\texttt{erase}(\hat{e}) \\
&\texttt{erase}(e_1\ e_2) = \texttt{erase}(e_1)\ \texttt{erase}(e_2) \\
&\texttt{erase}(e_1.\pi) = \texttt{erase}(e_1).\pi
\end{aligned}$$

Figure 3.12: Definitions of `annot` and `erase`.

restriction from OC where substitution is only well-defined when a variable is replaced with a value.

The multi-step rules of CC are identical to OC. Every single-step rule in OC is also a single-step rule in CC. The only difference is that the new annotated notation in CC would have a hat above every expression and type. For brevity, we omit those rules which are (basically) identical to those of OC.

The two new single-step reductions are given in 3.14. Both reduce `import` expressions. E-IMPORT1 reduces the definition of the capability being imported into the unannotated code. The more interesting rule is E-IMPORT2, which applies when the capability being impoted is a value. The unlabelled body e is annotated with the authority ε it selects; this is $\texttt{annot}(e, \varepsilon)$. The name x of the capability is then replaced with its actual definition $\hat{v}$; this is $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$. This reduction incurs no effects.

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

$[\hat{v}/y]x = \hat{v}$, if $x = y$
$[\hat{v}/y]x = x$, if $x \neq y$
$[\hat{v}/y](\lambda x : \hat{\tau}.\hat{e}) = \lambda x : \hat{\tau}.[\hat{v}/y]\hat{e}$, if $y \neq x$ and y does not occur free in $\hat{e}$
$[\hat{v}/y](\hat{e}_1\ \hat{e}_2) = ([\hat{v}/y]\hat{e}_1)([\hat{v}/y]\hat{e}_2)$
$[\hat{v}/y](\hat{e}_1.\pi) = ([\hat{v}/y]e_1).\pi$
$[\hat{v}/y](\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e) = \texttt{import}(\varepsilon)\ x = [\hat{v}/y]\hat{e}\ \texttt{in}\ e$

Figure 3.13: Definition of `substitution`.

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

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

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

Figure 3.14: New single-step reductions in CC.

3.2.3 CC Static Rules

Our goal in this section is to introduce ε-IMPORT, which approximates the effects of unannotated code by inspecting its selected authority. The rule is complicated, so we build up to it.

First, since programs in CC can be annotated or unannotated, we need to be able to recognise when either kind is well-typed. Since the annotated subset of CC contains OC, all the OC still apply, but the notation is different: 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 this change in notation the rules are the same, so we shall not repeat them.

The rules for typing unannotated pieces of code 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 3.15; each is analogous to some rule in OC, but the parts relating to effects have been removed.

Again, there are no rules which directly approximate the effects of unannotated code. The only mechanism for doing this is to encapsulate that code in an `import` expression, which restricts what authority it can exercise. This means the rules can only tell us something interesting about annotated code and unannotated code which is nested inside an `import` expression. For the rest of this section, we are going to build up to the definition of ε-IMPORT.

To begin, typing $\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e$ requires us to know that the capability $\hat{e}$ being imported is well-typed, so we add the premise $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1$. Whatever authority

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

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

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

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

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

Figure 3.15: (Sub)typing judgements for the unannotated sublanguage of CC

is exercised by e must have been selected by the `import` expression, so you should be able to type e with a binding for $x : \hat{\tau}$. However, $\hat{\tau}$ is annotated and e is unannotated, so we erase the labels on it. This gives our second premise, $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$. These observations give our first definition of ε-IMPORT in Figure 3.16. Since E-IMPORT2 labels the unannotated code code with the selected authority ε, an `import` expression should type to $\texttt{annot}(\tau, \varepsilon)$. The safe approximation of the unannotated code's effects is $\varepsilon_1 \cup \varepsilon$; the former comes from reducing the imported capability (which happens prior to the execution of the unannotated code) and the latter contains all the authority which the unannotated code may use.

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

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

Figure 3.16: A first rule for type-and-effect checking `import` expressions.

On the surface, this rule may seem overly restrictive because `import` only allows one to import a single capability. What about unannotated code which uses multiple capabilities? One solution is to reformulate the rules (and grammar) to permit any number of imports. Another is to import multiple capabilities by importing a tuple of capabilities; for example $\texttt{import}(\varepsilon)\ x = (\texttt{File}, \texttt{Socket})\ \texttt{in}\ e$. We have not presented tuples as a part of the base language, but they could be encoded as a derived form, or added as a language extension. Both approaches are straightforward and extending CC in these ways would be routine — but only allowing the import of a single capabilitiy reduces clutter in the rules and simplifies the presentation, so we stick to this convention.

The first version of the rule has some issues. First of all, there is no formal relation between ε and whatever effects are captured by $\hat{\tau}$. Consider $\hat{e} = \texttt{import}(\varnothing)\ x =$

`File in x.write`, which imports a `File` and writes to it, but declares its authority as $\varnothing$. According to ε-IMPORT1, $\vdash \hat{e}$: `Unit with` $\varnothing$, but this is clearly wrong. We need to add a constraint to say that whatever effects are captured by $\hat{\tau}$ should be selected 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 3.17. With it, we can add an extra premise $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ which formalises the idea that any capability exercised by e must be selected in ε. The updated rule is given in Figure 3.18.

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

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

Figure 3.17: A first definition of `effects`.

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

Figure 3.18: A second rule for type-and-effect checking `import` expressions.

The counterexample which defeated ε-IMPORT1 is now rejected by ε-IMPORT2, but the rule is not yet perfect: the annotations on one import can be broken by another import. To illustrate, consider an example where two capabilities are imported[1] in Figure 3.19. This program imports a function `go` which, when given a `Unit` $\to_\varnothing$ `Unit` function with no effects, will execute it. The other capability given is a `File`. The unannotated code creates a `Unit` $\to$ `Unit` function which writes to a file when executed, and passes it to `go`, which subsequently incurs the `File.write` effect.

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

Figure 3.19: Permitting multiple imports will break ε-IMPORT2.

In the world of annotated code, you cannot pass a file-writing function to `go`. However, typechecking of the unannotated body disregards the annotations of its imports, and since the file-writing function takes `unit` and returns `unit`, the type system will accept this program. This example shows that unannotated code might not use its capabilities in a way which respects their annotations. Because the unannotated code selects

[1] As stated before, we only formalise the case where code imports one capability, but this particular issue only arises with multiple imports.

$\{\texttt{File}.*\}$, the approximation on this `import` is actually safe at the top-level, but it contains locally unsafe code. We want to prevent this.

If `go` had the type $\texttt{Unit} \to_{\{\texttt{File.write}\}} \texttt{Unit}$ then the above example would be safely rejected. However, a modified version where a file-reading function is passed to `go` would still have the same issue. `go` is only safe when it expects every possible effect that the unannotated code could pass into it. For example, if `go` had the type $\texttt{Unit} \to_{\{\texttt{File}.*\}} \texttt{Unit}$, then the unannotated code cannot pass it a capability it isn't already expecting, and so it cannot violate the annotation on `go`. To solve the issue, we are going to require imported capabilities to have authority over ε, which contains every possible effect the unannotated code might use to violate their annotations.

To achieve greater control over this, we split the definition of `effects` into two separate functions, called `effects` and `ho-effects`. Definitions are given in Figure 3.20. The difference between the two is the difference between direct and transitive authority. If values of type $\hat{\tau}$ can directly incur an effect $r.\pi$, then $r.\pi \in \texttt{effects}(\hat{\tau})$. If values of type $\hat{\tau}$ can indirectly incur an effect $r.\pi$ — perhaps because they are given a capability for this effect as a function argument — then $r.\pi \in \texttt{ho-effects}(\hat{\tau})$. Put another way, if values of $\hat{\tau}$ possess a capability for that effect, then the capability's effects are contained in $\texttt{effects}(\hat{\tau})$. If values of $\hat{\tau}$ can incur an effect, but need to be given the capability by someone else in order to do that, then the effect belongs in $\texttt{ho-effects}(\hat{\tau})$.

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

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

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

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

Figure 3.20: Effect functions.

Note how the functions are mutually recursive, and have 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 $\texttt{ho-effects}(\{\bar{r}\}) = \varnothing$. There is also a correspondence between these definitions and the subtyping rule for 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 would still require $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ — unannotated code must select any capability which could be given to it — but a new premise $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau})$ would be added to formalise the idea that imported capabilities must know about every effect they could be given by unannotated code. The counterexample

from Figure 3.19 will now be rejected, because $\texttt{ho-effects}(\texttt{Unit} \to_\varnothing \texttt{Unit}) \to_\varnothing \texttt{Unit}) = \varnothing$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$. But this is *still* not sufficient! Consider $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2)$. We want *every* higher-order capability involved to be expecting ε, but $\subseteq$ does not distribute over $\to_{\varepsilon'}$. 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 solution is to define a relation like $\varepsilon \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$, but where the $\subseteq$ distributes over the input and output type. The two relations are called `safe` and `ho-safe`. The former is a distributive version of $\varepsilon \subseteq \texttt{effects}(\hat{\tau})$ and the latter of $\varepsilon \subseteq \texttt{ho-effects}(\hat{\tau})$. Definitions are given in 3.21.

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

$$\frac{}{\texttt{safe}(\{\bar{r}\}, \varepsilon)}\ (\textsc{Safe-Resource}) \qquad \frac{}{\texttt{safe}(\texttt{Unit}, \varepsilon)}\ (\textsc{Safe-Unit})$$

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

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

$$\frac{}{\texttt{ho-safe}(\{\bar{r}\}, \varepsilon)}\ (\textsc{HOSafe-Resource}) \qquad \frac{}{\texttt{ho-safe}(\texttt{Unit}, \varepsilon)}\ (\textsc{HOSafe-Unit})$$

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

Figure 3.21: Safety judgements in the epsilon calculus.

Note again how the mutual recursion of `safe` and `ho-safe` mimics the co(ntra)variance rules for function subtyping. Some properties of these relations 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, so the safety judgements are stronger notions.

We are now ready to give an amended version of ε-IMPORT. The rule is stated in Figure 3.22, and 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 constitutent piece of $\hat{\tau}$ — must be expecting the effects which might be passed to it inside the unannotated code, which are contained in ε.

The premises so far are comprehensively strict about what capabilities might be directly exercised by the unannotated code — but what about those which are indirectly exercised? Consider $\hat{e} = \texttt{import}(\varnothing)\ x = \texttt{unit}\ \texttt{in}\ \lambda \texttt{f} : \texttt{File}.\ \texttt{f.write}$. The unannotated selects

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

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

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

no capabilities and returns a function which, when given a `File`, will incur `File.write`. This satisfies the premises in ε-IMPORT3, but the ascribed type is $\{\texttt{File}\} \to_{\varnothing} \texttt{Unit}$ — not good!

So far we have only focused on the capabilities ε which unannotated code might select and be endowed with, but a capability might also be introduced to unannotated code during runtime by being passed in as a function argument. Suppose the unannotated code returns a function f. Then f may take as input either a resource or a function. If a function, then the permitted effects of this function are bound by ε, because the `import` annotates f with ε. Trying to invoke $\texttt{annot}(f, \varepsilon)$ back in annotated code with a capability whose authority exceeds ε would therefore be rejected by the soundness of `OC`, so we are automatically higher-order safe in this case. However, if a resource is given to f then it could directly invoke any effect on f without having selected it in ε. To prevent this, we add the premise $\texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon)) \subseteq \varepsilon$. This will collect every effect captured by a resource literal passed into the unannotated code and ensure it has been selected. `hofx` is only defined on annotated types, so we must first annotate τ in order to apply the function — the annotated type is otherwise not relevant to this situation.

The final version of ε-IMPORT is given in Figure 3.23.

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

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

Figure 3.23: The final rule for typing imports.

3.2.4 `CC` Soundness

Annotated programs are the only ones which can be reduced and have their effects approximated, so the statement of soundness in `CC` only applies to those situations. A definition is given below.

Theorem 11 (Soundness). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A *and* $\hat{e}_A$ *is not a value, then* $e_A \longrightarrow e_B \mid \varepsilon$, *where* $\hat{\Gamma} \vdash e_B : \hat{\tau}_B$ `with` ε_B *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

Since the rules of OC are also contained in CC and have been proven sound, we shall present the same theorems in this section, but only comment on the cases which pertain to new forms and rules. Some lemmas are new, and in those cases we prove them for every case.

We begin with canonical forms, which remains unchanged. The substitution lemma gains an extra case, but the proof is routine.

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

- *If* $\Gamma \vdash v : \tau$ `with` ε *then* $\varepsilon = \varnothing$.
- *If* $\Gamma \vdash v : \{\bar{r}\}$ *then* $v = r$ *for some* $r \in R$ *and* $\{\bar{r}\} = \{r\}$.

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

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

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

Similarly, the progress theorem now has an extra case for when the typing rule used was ε-IMPORT. The proof is also straightforward.

Theorem 12 (Progress). *If* $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ `with` ε *and* $\hat{e}$ *is not a value, then* $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon$.

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

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

The preservation theorem also gains an extra case for when ε-IMPORT is the typing rule used. The subcase where E-IMPORT1 is the reduction rule used is straightforward, but the other subcase where E-IMPORT2 is used is tricky. We devote most of the rest of this section to building up the appropriate lemmas needed to show that this particular subcase is type-and-effect safe.

In this particular situation the capability being imported is a value and the reduction annotates the naked code with its selecetd authority ε and then performs substitution. The reduction has the form $\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon) \mid \varnothing$. To show it preserves type-and-effect safety requires a few things. First, if $\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e : \hat{\tau}_A$ `with` ε, then we need to be able to type the reduced expression in the same context: $\hat{\Gamma} \vdash [\hat{v}/x]\texttt{annot}(e, \varepsilon) : \hat{\tau}_B$ `with` ε_B. Since $\hat{v}$ is a value then by the conclusion of ε-IMPORT, $\varepsilon_B = \varepsilon$, the selected authority of e. Showing $\varepsilon \subseteq \varepsilon_A$ would be efficient to show effect-soundness.

To show type-soundness we need $\tau_B <: \tau_A$. But what is τ_B? Intuitively, if $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$, and the expression after reduction is $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$, then the type after reduction is probably related to τ in some way. By substitution lemma, the type of $\texttt{annot}(e, \varepsilon)$ ought to be preserved after substitution. So we might guess the type will be either $\texttt{annot}(\tau, \varepsilon)$ or $\texttt{annot}(\hat{\tau}, \varepsilon)$. The first of those two is what we prove with the following lemma.

Lemma 7 (Annotation)**.** *If the following are true:*

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

Let $\varepsilon' = \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$. *Then* $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon'), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon') : \texttt{annot}(\tau, \varepsilon')$ `with` ε'.

The premises of the annotation lemma are very specific to the premises of ε-IMPORT, but generalised slightly to accommodate a proof by induction: note how there are ambient bindings in the typechecking of unannotated code: $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e : \tau$. Furthermore, the approximated effects in the judgement in the conclusion of the theorem statement contain everything captured by these ambient bindings: $\texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$, which is essentially the ambient authority exercised in e. By contrast, ε is the selected authority of e.

Because of the constraints imposed on e by ε-IMPORT, no effect at the top-level is ambient.; therefore when we apply the annotation lemma we choose $\Gamma = \varnothing$. However, inductively-speaking, some effects are ambient in certain sub-scopes of e. For example, the expression `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 considered ambient in the enclosing scope. In the process of proving the lemma we may need to, for example, step into the body of a function, at which point certain capabilities become ambient and need to be considered as such, and so we need to consider at all points what is the ambient authority.

Note that when $\Gamma = \varnothing$ then $\varepsilon' = \varepsilon$ and applying the lemma gives a typing judgement of the form $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon'), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon') : \texttt{annot}(\tau, \varepsilon')$ `with` ε, which matches the judgement in the conclusion of ε-IMPORT, before the substitution $[\hat{v}/x]$ is made.

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

Lemma 8. *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 9. *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 `ho-safe` and `safe`. □

Again, there is a close relation between these lemmas and the subtyping rule for functions. In a subtyping relation between functions, the input type is contravariant. Therefore, if $\hat{\tau} = \hat{\tau}_1 \to_{\varepsilon'} \tau_2$ and we have $\hat{\tau} <: \texttt{annot}(\tau, \varepsilon)$, then we need to know $\texttt{annot}(\tau_1) <: \hat{\tau}_1$. This is why there are two lemmas: one for each direction.

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

Theorem 13 (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* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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

We can now combine Progress and Preservation into the Soundness theorem for CC. The proof of multi-step soundness in CC is identical to the proof in OC.

Theorem 14 (Soundness). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A$ *is not a value, then* $e_A \longrightarrow e_B \mid \varepsilon$, *where* $\hat{\Gamma} \vdash e_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

Theorem 15 (Multi-step Soundness). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $e_A \longrightarrow^* e_B \mid \varepsilon$, *where* $\hat{\Gamma} \vdash e_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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 $\texttt{unit} \stackrel{\text{def}}{=} \lambda x : \varnothing.x$. The `unit` literal is the same in both annotated and naked code. In annotated code, it has the type $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to_{\varnothing} \varnothing$, while in naked code it has the type $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to \varnothing$. While these are technically two 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, $\vdash$ `unit` : `Unit with` $\varnothing$ by ε-ABS, and $\vdash$ `unit` : `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`.

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

```
1 λx:Unit. unit
```

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

4.1.2 Let

The expression `let` $x = \hat{e}_1$ `in` $\hat{e}_2$ first binds 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 `let` $x = \hat{e}_1$ `in` $\hat{e}_2 \stackrel{\text{def}}{=} (\lambda x : \hat{\tau}_1.\hat{e}_2)\hat{e}_1$. 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)

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

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

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

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

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

Figure 4.3: Derived `let` rules.

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.

```
1 require File
2 instantiate PureMod
3 instantiate ResourceMod(File)
4
5 def main():Unit
6   PureMod.tick(File)
7   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 correspond-

ing 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.

```
let MakeLogger =
  (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 $\texttt{f} : \{\texttt{File}\} \vdash \texttt{LoggerModule} : \texttt{Unit} \to_{\texttt{File.write}} \texttt{Unit}\ \texttt{with}\ \varnothing$. At line 13 the same rule gives $\texttt{f} : \{\texttt{File}\} \vdash \texttt{ClientModule} : (\texttt{Unit} \to_{\varnothing} \texttt{Unit}) \to \texttt{Unit}\ \texttt{with}\ \varnothing$. 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()
```

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

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

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

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

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 {`File`}, and `effects`({`File`}) = {`File`.$*$}, which is inconsistent with the declared effects $\varnothing$.

```
let MakeLogger =
```

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

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

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

(MakeMain File) unit
```

The only way for this to typecheck would be to annotate `Client.run` as having every effect on `File`. This demonstrates how the effect-system of `CC` approximates 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 λ`f` : `File.f.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 `y` = `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.

```
module Malicious

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

```
module Plugin
instantiate Malicious

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

```
resource module Main
require File
instantiate Plugin

Plugin.run(File)
```

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

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

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

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

4.2.6 Resource Leak

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

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

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

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

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

```
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 $\texttt{import}(\texttt{File}.*)\ x = unit\ \texttt{in}\ \lambda f : \{\texttt{File}\}.\ f.write$ 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 $\lambda x : \texttt{Unit} \to_\varepsilon \texttt{Unit}.\ x\ \texttt{unit}$, 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 (Canonical Forms)**.** *Unless the rule used is* ε-SUBSUME*, the following are true:*

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

Proof.

1. 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, $\varepsilon = \varnothing$.
2. The only rule ascribing the type $\{\bar{r}\}$ is ε-RESOURCE. Its premises imply the result.
3. The only rule ascribing the type $\tau_1 \to_{\varepsilon'} \tau_2$ is ε-ABS. Its premises imply the result.

□

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

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

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

Case: ε-APP. Then $e = e_1\ e_2$. If e_1 is a non-value it can be reduced by inductive assumption, and then $e_1\ e_2 \longrightarrow e_1'\ e_2$ by E-APP1. If $e_1 = v_1$ is a value and e_2 a non-value, then e_2 can be reduced by inductive assumption and then $e_1\ e_2 \longrightarrow v_1\ e_2'$ by E-APP2. Otherwise $e_1 = v_1$ and $e_2 = v_2$ are both values. By canonical forms, $v_1 = \lambda x : \tau.e$. Then $(\lambda x : \tau.e)v_2 \longrightarrow [v_2/x] \mid \varnothing$ by E-APP3.

Case: ε-OPER. Then $e = e_1.\pi$. If e_1 is a non-value it can be reduced by inductive assumption, and then $e_1.\pi \longrightarrow e_1'.\pi \mid \varepsilon_1$ by E-OPERCALL1. Otherwise $e_1 = v_1$ is a value. By

canonical forms, $v_1 = r$. Then $r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}$ by E-OperCall2.

Case: ε-Subsume. Then $\Gamma \vdash e : \tau' \texttt{ with } \varepsilon'$. By inversion, $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$, where $\tau' <: \tau$ and $\varepsilon' \subseteq \varepsilon$. If e is a value or variable, the theorem holds vacuously. Otherwise the reduction exists by applying the inductive assumption to the sub-derivation. □

Lemma 11 (Substitution). *If* $\Gamma, x : \tau' \vdash e : \tau \texttt{ with } \varepsilon$ *and* $\Gamma \vdash v : \tau' \texttt{ with } \varnothing$ *then* $\Gamma \vdash [v/x]e : \tau \texttt{ with } \varepsilon$.

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

Case: ε-Var. Then $e = y$ and either $y = x$ or $y \neq x$.

Subcase 1: $y \neq x$. By theorem assumption, $\Gamma \vdash y : \tau \texttt{ with } \varepsilon$, where $\varepsilon = \varnothing$ by inversion on ε-Var. Since $[v/x]y = y$, then $\Gamma \vdash [v/x]y : \tau \texttt{ with } \varnothing$.

Subcase 2: $y = x$. By inversion on ε-Var, the typing judgement from the theorem assumption is $\Gamma, x : \tau' \vdash x : \tau' \texttt{ with } \varnothing$. Since $[v/x]y = v$, and by assumption $\Gamma \vdash v : \tau' \texttt{ with } \varnothing$, then $\Gamma \vdash [v/x]x : \tau' \texttt{ with } \varnothing$.

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

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

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

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

□

Theorem 17 (Preservation). *If* $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ *and* $e_A \longrightarrow e_B \mid \varepsilon_C$, *then* $\Gamma \vdash e_B : \tau_B \texttt{ with } \varepsilon_B$, *where* $e_B <: e_A$ *and* $\varepsilon \cup \varepsilon_B \subseteq \varepsilon_A$.

Proof. By induction on $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$, and then on $e_A \longrightarrow e_B \mid \varepsilon$.

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 $e_1 : \tau_2 \longrightarrow_\varepsilon \tau_3 \texttt{ with } \varepsilon_1$ and $\Gamma \vdash e_2 : \tau_2 \texttt{ with } \varepsilon_2$ and $\tau_B = \tau_3$ and $\varepsilon_A = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon$.

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

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

Subcase: E-APP3. Then $e_1 = \lambda x : \tau_2.e$ and $e_2 = v_2$ are values, and $(\lambda x : \tau_2.e)\ v_2 \longrightarrow [v_2/x]e \mid \varnothing$. By inversion on the typing rule for $\lambda x : \tau_2.e$, we know $\Gamma, x : \tau_2 \vdash e : \tau_3 \texttt{ with } \varepsilon_3$. By canonical forms, $\varepsilon_2 = \varnothing$ because $e_2 = v_2$ is a value. Then by the substitution lemma, $\Gamma \vdash [v_2/x]e : \tau_3 \texttt{ with } \varepsilon_3$. 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 $e_1 : \{\bar{r}\} \texttt{ with } \varepsilon_1$ and $\tau_A = \texttt{Unit}$ and $\varepsilon_A = \varepsilon_1 \cup \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$.

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

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

Theorem 18 (Soundness). *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 $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.*

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

Theorem 19 (Multi-step Soundness). *If $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ and $e_A \longrightarrow^* e_B \mid \varepsilon$, where $\Gamma \vdash e_B : \tau_B \texttt{ with } \varepsilon_B$ and $\tau_B <: \tau_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.*

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

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

Case: Length 1. Then the result follows by single-step soundness.

Case: Length $n + 1$. Then 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 $\Gamma \vdash e_B : \tau_B \texttt{ with } \varepsilon_B$. By inductive assumption, $\tau_C <: \tau_A$ 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

Theorem 20 (Progress). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}$ `with` ε *and* $\hat{e}_A$ *is not a value or variable, then* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$.

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

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

Lemma 12 (Substitution). *If* $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A *and* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}'$ `with` $\varnothing$ *then* $\hat{\Gamma} \vdash [\hat{v}/x]e : \hat{\tau}_A$ `with` ε_A.

Proof. By induction on $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A.

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

1. $\hat{\Gamma}, x : \hat{\tau}' \vdash \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)$ `with` $\varepsilon \cup \varepsilon_1$
2. $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}$ `with` ε_1
3. $\varepsilon = \texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon))$
4. $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$
5. $\hat{\tau}_A = \texttt{annot}(\tau, \varepsilon)$
6. $\hat{\varepsilon}_A = \varepsilon \cup \varepsilon_1$

By applying inductive assumption to (2), $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}$ `with` ε_1. Then by ε-MODULE, $\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = [\hat{v}/x]\hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon)$ `with` $\varepsilon \cup \varepsilon_1$. □

Lemma 13. *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 14. *If* $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ *and* $\texttt{safe}(\hat{\tau}, \varepsilon)$ *then* $\texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon) <: \hat{\tau}$.

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

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

Case: $\hat{\tau} = \hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2$, $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ It is sufficient to show $\hat{\tau}_2 <: \texttt{annot}(\texttt{erase}(\hat{\tau}_2), \varepsilon)$ and $\texttt{annot}(\texttt{erase}(\hat{\tau}_1), \varepsilon) <: \hat{\tau}_1$, because the result will hold by S-EFFECTS. To achieve this we shall inductively apply **lemma 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: $\hat{\tau} = \hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2$, $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{safe}(\hat{\tau}, \varepsilon)$ It is sufficient to show $\texttt{annot}(\texttt{erase}(\hat{\tau}_2), \varepsilon) <: \hat{\tau}_2$ and $\hat{\tau}_1 <: \texttt{annot}(\texttt{erase}(\hat{\tau}_1), \varepsilon)$, because the result will hold by S-EFFECTS. To achieve this we shall inductively apply **lemma 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 15 (Annotation)**.** *If the following are true:*

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

Let $\varepsilon' = \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$. *Then* $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon'), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon') : \texttt{annot}(\tau, \varepsilon')$ `with` ε'.

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

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

Subcase 1: $x = y$. Then by assumption, $y : \texttt{erase}(\hat{\tau}) \vdash x : \tau$, so $\tau = \texttt{erase}(\hat{\tau})$. By ε-VAR, $y : \hat{\tau} \vdash x : \hat{\tau}$ `with` $\varnothing$. By definition, $\texttt{annot}(x, \varepsilon') = x$, so $y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon') :$

$\hat{\tau}$ with $\varnothing$. By assumptions (3) and (4) we know $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ and $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$, so applying **Lemma 2** gives $\hat{\tau} <: \texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$. Then applying ε-SUBSUME to the type of $y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon) : \hat{\tau}$ with $\varnothing$ gives $y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon) : \texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$ with $\varnothing$. We already know $\texttt{erase}(\hat{\tau}) = \tau$, so this judgement is the same as $y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon) : \texttt{annot}(\tau, \varepsilon)$ with $\varnothing$. By applying ε-SUBSUME to $\varnothing$ we get $y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon) : \texttt{annot}(\tau, \varepsilon)$ with $\varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$. Lastly, by widening the context, we have $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon) : \texttt{annot}(\tau, \varepsilon)$ with $\varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$.

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

Case: T-RESOURCE. Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash r : \{r\}$. By definition, $\texttt{annot}(r, \varepsilon) = r$ and $\texttt{annot}(\{r\}, \varepsilon)$. By ε-RESOURCE $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash r : \{r\}$ with $\varnothing$. By ε-SUBSUME, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash r : \{r\}$ with $\varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$.

Case: T-ABS. Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash \lambda x : \tau_1.e_{body} : \tau_1 \to \tau_2$. Applying definitions, $\texttt{annot}(e, \varepsilon') = \texttt{annot}(\lambda x : \tau_1.e_2, \varepsilon') = \lambda x : \texttt{annot}(\tau_1, \varepsilon').\texttt{annot}(e_2, \varepsilon')$ and $\texttt{annot}(\tau, \varepsilon') = \texttt{annot}(\tau_1 \to \tau_2, \varepsilon') = \texttt{annot}(\tau_1, \varepsilon') \to_{\varepsilon'} \texttt{annot}(\tau_2, \varepsilon')$. By inversion on T-ABS, we get the sub-derivation $\Gamma, y : \texttt{erase}(\hat{\tau}), x : \tau_1 \vdash e_{body} : \tau_2$. We shall apply the inductive assumption to this judgement with an unlabelled context $\Gamma, x : \tau_1$. Define $\varepsilon'' = \varepsilon' \cup \texttt{effects}(\texttt{annot}(\tau_1, \varepsilon))$. ε'' is the inductive assumption's version of ε'. Applying the inductive assumption, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon''), y : \hat{\tau}, x : \texttt{annot}(\tau_1, \varepsilon'') \vdash \texttt{annot}(e_{body}, \varepsilon'') : \texttt{annot}(\tau_2, \varepsilon'')$ with ε''.

Ostensibly, ε'' seems to differ from ε' by $\texttt{effects}(\texttt{annot}(\tau_1, \varepsilon'))$, but we shall show $\varepsilon' = \varepsilon''$. By assumption (3), $\varepsilon' = \texttt{effects}(\hat{\tau}) \cup \texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon))$. In this case, $\tau = \tau_1 \to \tau_2$, so $\texttt{annot}(\tau, \varepsilon) = \texttt{annot}(\tau_1, \varepsilon) \to_{\varepsilon} \texttt{annot}(\tau_2, \varepsilon)$. By definition, $\texttt{effects}(\texttt{annot}(\tau_1, \varepsilon)) \subseteq \texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon))$, so $\texttt{effects}(\texttt{annot}(\tau_1, \varepsilon)) \subseteq \varepsilon$. Therefore, $\varepsilon'' = \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$. But this is the definition of ε', so $\varepsilon'' = \varepsilon'$.

We can therefore rewrite the judgement obtained from the inductive hypothesis as $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon'), y : \hat{\tau}, x : \texttt{annot}(\tau_1, \varepsilon') \vdash \texttt{annot}(e_{body}, \varepsilon') : \texttt{annot}(\tau_2, \varepsilon')$ with ε'. Applying ε-ABS, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon'), y : \hat{\tau} \vdash \lambda x : \texttt{annot}(\tau_1, \varepsilon').\texttt{annot}(e_{body}, \varepsilon')$ with $\varnothing$. Lastly, by ε-SUBSUME, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\hat{\tau}_1) \to_{\varepsilon} \texttt{annot}(\hat{\tau}_2)$ with $\varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma), \varepsilon)$.

Case: T-APP Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1\ e_2 : \tau_3$, where $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1 : \tau_2 \to \tau_3$ and $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_2 : \tau_2$. By applying the inductive assumption to e_1 and e_2, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e_1, \varepsilon) : \texttt{annot}(\tau_1, \varepsilon)$ with ε and $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e_2, \varepsilon) : \texttt{annot}(\tau_2, \varepsilon)$ with ε. Simplifying, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e_1, \varepsilon) : \texttt{annot}(\tau_2, \varepsilon) \to_{\varepsilon} \texttt{annot}(\tau_3, \varepsilon)$ with ε. Then by ε-APP, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e_1\ e_2, \varepsilon) : \texttt{annot}(\tau_3, \varepsilon)$ with ε.

Case: T-OPERCALL Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1.\pi : \texttt{Unit}$. By inversion we get the subderivation $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1 : \{\bar{r}\}$. By definition, $\texttt{annot}(\{\bar{r}\}, \varepsilon) = \{\bar{r}\}$. By inductive assumption, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash e_1 : \{\bar{r}\}$ $\texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$. By ε-OPERCALL, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash e_1.\pi : \{\bar{r}\}$ $\texttt{with}\ \varepsilon \cup \{\bar{r}.\pi\}$.

It remains to show $\{\bar{r}.\pi\} \subseteq \varepsilon$. We shall do this by considering where r must have come from (which subcontext left of the turnstile).

Subcase 1. $r = \hat{\tau}$. As $\varepsilon = \texttt{effects}(\hat{\tau})$, then $r.\pi \in \texttt{effects}(\hat{\tau})$.

Subcase 2. $r : \{r\} \in \Gamma$. As $\texttt{annot}(r, \varepsilon) = r$, then $r.\pi \in \texttt{annot}(\Gamma, \varepsilon)$.

Subcase 3. $r : \{r\} \in \hat{\Gamma}$. Then because $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1 : \{\bar{r}\}$, then $r \in \Gamma$ or $r = \texttt{erase}(\hat{\tau}) = \hat{\tau}$ and one of the above subcases must also hold.

□

Theorem 21 (Preservation). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}\ \varepsilon_A$ *and* $e_A \longrightarrow e_B \mid \varepsilon_C$, *then* $\hat{\Gamma} \vdash e_B : \tau_B$ $\texttt{with}\ \varepsilon_B$, *where* $e_B <: e_A$ *and* $\varepsilon \cup \varepsilon_B \subseteq \varepsilon_A$.

Proof. By induction on $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}\ \varepsilon_A$, and then on $e_A \longrightarrow e_B \mid \varepsilon$.

Case: ε-MODULE Then $e_A = \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e$.

Subcase: E-MODULE1 If the reduction rule used was E-MODULECALL1 then the result follows by applying the inductive hypothesis to $\hat{e}$.

Subcase: E-MODULE2 Otherwise $\hat{e}$ is a value and the reduction used was E-MODULECALL2. The following are true:

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

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

□

Theorem 22 (Soundness). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A$ *is not a value, then* $e_A \longrightarrow e_B \mid \varepsilon$, *where* $\hat{\Gamma} \vdash e_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

Theorem 23 (Multi-step Soundness). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A \longrightarrow^* e_B \mid \varepsilon$, *where* $\hat{\Gamma} \vdash e_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$ *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

Proof. The proofs are 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] MAASS, M. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016.

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

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

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

[15] 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.

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

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

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

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

[20] 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.

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

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

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

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