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

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 help achieve these qualities.

It is difficult to determine if a piece of code is trustworthy. Several scenarios can give rise to the issue of trust, such as in a development environment adhering to *code ownership*. In this setting, groups of developers may function as experts over certain components. When their components must interact with code sourced from outside their domain of expertise, they can make false assumptions or violate the internal constraints of other components by using them incorrectly. Another setting involves applications which allow third-party plug-ins, in which case third-party code is sourced from an untrustworthy source. A web mash-up is a particular kind of software that brings together disparate applications into a central service, in which case the disparate applications may be untrustworthy.

A range of methods might be employed to help us determine whether we can trust a piece of code. Sandboxing is one, where the suspect code is executed in a safe, separate environment from the trusted code, but this approach has many shortcomings [**?**]. Verification techniques allow for robust analyses on code, but are heavyweight and require the developers to have a deep understanding of the techniques being employed [**?**]. Lightweight analyses, such as those present in type system, enjoy the benefit of being easy for the developer to use, but existing languages do not provide adequate controls for detecting and isolating untrustworthy components [**?**].

A qualitative approach to trustworthiness is to develop according to particular guidelines considered to be in good practice. 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 [15]. For example, a logger module, which need only 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 [**?**].

This report is interested in how type systems might enforce more static constraints, putting developers in a more-informed position to make qualitative judgements about the trustworthiness of code.

One approach to privilege separation is the *capability* model. A capability is an unforgeable token granting its bearer permission to perform some operation [4]. Resources in a program are only exercised though the capabilities granting them. Although the notion of a capability is an old one, 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" [11]. In his paradigm, the reference graph of a program is the same as the access graph. This eliminates *ambient authority*, whereby a privilege is exercised without being explicitly declared. This enables one to reason about what privileges a component might exercise by examining its interface. Building on these ideas, Maffeis et. al. formalised *capability-safety* of a language, showing a subset of Caja (a JavaScript implementation) meets this notion [9].

In addition to realising privilege separation, capabilities also encapsulate the source of *effects*. An *effect* describes some intensional information about the way in which a program executes [12]. For example, a logger's method might `append` to a file, and so executing this method would incur {`File.append`}. To be able to do this in a capability-safe language, the logger must have a capability for the `File`. Therefore, the constraints imposed by how capabilities proliferate, also impose constraints on what effects the components of a program may incur.

Capabilities can offer fine-grained control over the way in which privileges are exercised via *confinement*. For example, while we expect the logger's log method to have the `append` effect, a sloppy or malicious implementation may incur extra effects on the `File`, such as `write` or `close`. Knowing the logger might incur these extra effects may inform the decision a developer makes about whether or not to trust this particular component.

An effect-system is an extension to a type-system, which track what effects a program might incur, and where. They have been used to do **this, this, and that**. Some have criticised their verbosity **lightweight polymorphic effects paper** as a point against their practical adoption. An effect system such as the Talpin-Jouvelot `ETL` system **ATAPL?** requires the annotation of all values in a program. This requires the developer to be aware, at all points, of what effects are in scope. **Is this true of all, or even most, effect systems?** Minor alterations to the signatures and effects of one component might require the labels on all interacting components to change in accordance. This overhead is something the developer must carry with them at all stages of development, affecting their usability in large systems. Successive works have focussed on reducing these issues through techniques such as effect-inference, but the benefit of capabilities for effect-based reasoning has received less attention.

Because capabilities encapsulate effectful privileges, and because capability-safe languages impose constraints on how these privileges can spread throughout the system, this considerably simplifies effect-based reasoning. To incur an effect requires one to possess a capability for the appropriate resource, and whether this resource is captured by a component can be determined by inspecting the type-signatures of the component. The developer need not look at the source code. This is the key contribution of this report: that capability-safety facilitates a low-cost effect-based reasoning with minimal user overhead.

We begin this paper by discussing preliminary concepts involving the formal definition of programming languages (2.1.), effect systems (2.2.), and the capability model (2.3.). Along the way we summarise some existing languages to illustrate these points.

Chapter 3 introduces a pair of languages called $\lambda^{\rightarrow}_{\pi}$ and $\lambda^{\rightarrow}_{\pi,\varepsilon}$. $\lambda^{\rightarrow}_{\pi}$ is a typed lambda calculus with a simple notion of capabilities and runtime effects. Every function in $\lambda^{\rightarrow}_{\pi}$ is annotated, which gives a simple, sound system for determining what effects a piece of code might incur. $\lambda^{\rightarrow}_{\pi,\varepsilon}$ allows for unannotated code by introducing an `import` construct. At the point of interaction between labelled and unlabelled code, a capability-based reasoning enables us to make a safe inference about what effects the unlabelled code might incur.

Chapter 4 shows how $\lambda^{\rightarrow}_{\pi,\varepsilon}$ might be used practically, and we try to convince the reader that $\lambda^{\rightarrow}_{\pi,\varepsilon}$ can be implemented in existing capability-safe languages in a routine manner. We finish with a literature comparison.

Chapter 2

Background

In this section we cover some of the necessary concepts and existing work informing this report. First we cover the process of formally defining a programming language and proving some of its properties. For this purpose, we present EBL. We then summarise the rules and properties of a variation of the simply-typed lambda calculus $\lambda^{\rightarrow}$. $\lambda^{\rightarrow}$ is an historically important model of computation and serves as a basis for many functional programming language. It is also the basis of $\lambda^{\rightarrow}_{\pi,\varepsilon}$, so a preliminary understanding of $\lambda^{\rightarrow}$ will help us understand $\lambda^{\rightarrow}_{\pi,\varepsilon}$.

$\lambda^{\rightarrow}_{\pi,\varepsilon}$ is a capability-based language with an effect system. To understand what that 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 formally by supplying three sets of rules: a grammar, which defines syntactically legal terms; static rules, which determine whether programs meet certain well-formedness properties; anddynamic rules, which express the meaning of a program by defining how they are executed. When a language has been defined we want to know its rules are mathematically correct.

We illustrate these concepts by presenting EBL, which is a simple, type-safe language based around boolean and arithmetic expressions. Like every language in this report, it is expression-based, meaning that valid programs can always be evaluated to yield a value. Although EBL is not very interesting, the process of defining it and proving its rules correct illustrate the general approach this report will take towards formally specifying programming languages.

2.1.1 Grammar

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

A `EBL` program is an expression e, consisting of variable definitions and the application of boolean and arithmetic operations. A valid expression is either a variable, a constant (such as 3, 0, `true`, or `false`), by joining two other valid expressions using $+$ or $\vee$, or by introducing a binding for a variable in a piece of code (`let` expression). The following are examples of syntactically legal programs: x, y, 3, $3+2$, $\texttt{false} \vee \texttt{true}$, $3 \vee \texttt{false}$, $\texttt{true} + \texttt{false}$, $\texttt{let}\ x = 3\ \texttt{in}\ x + 1$, $3 + (x + 2)$.

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. **A diagram might be nice here**. Sometimes 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 is an implementation detail, so throughout this report we only consider strings which unambiguously correspond to a valid AST.

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

Figure 2.1: Grammar for `EBL` expressions.

2.1.2 Dynamic Rules

The dynamic rules of a language specify the meaning of syntactically-valid terms. There are different approaches, but the one we use is called *small-step semantics*, where the meaning of a program is given by how it is executed. This is specified as a set of *inference rules*. An inference rule is given as a set of premises above a dividing line which, if they hold, imply the result below the line. If an inference rule has no premises it is called an *axiom*. An instantiation of a particular inference rule is called a judgement.

$\boxed{e \longrightarrow e}$

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

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

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

Figure 2.2: Inference rules for single-step reductions.

Figure 2.4. gives the dynamic rules for EBL. Their conclusions specify members of 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'.

If a non-variable expression is irreducible under the dynamic rules, it is called a value. The grammar of EBL also specifies a category of terms called "value". As we shall see, these two definitions correspond. 3, `true`, and `false` are examples of irreducible expressions.

A disjunction is reduced by first reducing the left-hand side to a value (E-OR1). If the left-hand side is the boolean literal `true`, then we can reduce the expression to `true` (because $\texttt{true} \vee Q = \texttt{true}$). Otherwise if the left-hand side is the boolean literal `false`, we can reduce the expression to the right-hand side e_2 (because $\texttt{false} \vee Q = Q$). This particular formulation of the rules encodes short-circuiting behaviour into $\vee$, meaning that if the left-hand side is true, the expression evaluates to true without checking the right-hand side.

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

A let expression is reduced by first reducing the subexpression being bound (E-LET1). When that is a value, we substitute the variable x for the value 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 an application of E-LET2.

Consider $\texttt{let}\ x = 1 + 1\ \texttt{in}\ x + 1$. According to the rules, $1 + 1$ would first be reduced to 2 before the substitution is made into $x + 1$. This strategy of reducing expressions before they are bound to variable names is *call-by-value*.

Formally, substitution is a function operating on expressions. A definition is given in

Figure 2.5. The notation $[e_1/x]e$ is short-hand for $\texttt{substitution}(e, e_1, x)$. When performing 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 as they are written, left-to-right.

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

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

Figure 2.3: Substitution for EBL.

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

To illustrate the solution, consider $\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ x = 2\ \texttt{in}\ x + z)$. In some sense, this is an equivalent program to $\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ y = 2\ \texttt{in}\ y + z)$. Because the names of variables are arbitrary, 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 [14, 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.

Given a single-step reduction relation, we may 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. shows how multi-step reduction can be defined with a set of inference rules.

[1]We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation [14, p. 39].

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

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

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

Figure 2.4: Dynamic rules.

2.1.3 Static Rules

If you try to reduce some terms you either end up with nonsense or get stuck in a situation where no rule applies. For example, $(1 + 1) + \texttt{false} \longrightarrow 2 + \texttt{false}$ by E-ADD1, but then you are stuck. $\texttt{false} \vee 3 \longrightarrow 3$ by E-OR3, which is strange.

When designing a language we often want to consider those syntactically legal terms satisfying 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 useful well-formedness property says that every variable used in a program must be declared beforehand. The examples above are *not* well-typed, because they are applying operators to arguments of the wrong type. We want rules to help us determine if this is the case without having to execute the program.

The static rules of EBL describe a basic type system which let us determine, without executing a program, whether it contains type errors. The relevant constructs for EBL are given as a grammar in Figure 2.5. There are two types: Nat and Bool. Furthermore, there is the notion of a *typing context*, which maps variables to their types. This is needed in the case of a program like $\texttt{let}\ x = 1\ \texttt{in}\ x + 1$. In trying to determine whether $x + 1$ is well-typed, we need to know what is the type of x. To do this, when we see the binding $x = 1$ we extend the context to say that x has type Nat.

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

Figure 2.5: Grammar for arithmetic expressions.

Figure 2.6. summarises the static rules of EBL. Note that every judgement holds in a particular typing context. For example, the judgement $x : \texttt{Int} \vdash x + 1 : \texttt{Int}$ is a claim about a particular property ($x + 1$ has the type Int) in a particular context (the context where x is an `Int`). If a judgement can be derived from the empty context, we conventionally write it as $\vdash e : \tau$ instead of $\varnothing \vdash e : \tau$.

T-BOOL and T-NAT are rules which say that constants always type to `Bool` or `Nat`. T-VAR says that a variable types to whatever the context binds it to. T-OR types a disjunction if the arguments are both `Bool`. T-ADD types a sum if the arguments are both `Nat`. The most interesting rule is T-LET, where the context gains a binding for x when type-checking the body of the `let` expression. The type of a `let` expression is the type of its body.

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

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

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

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

Figure 2.6: Inference rules for typing arithmetic expressions.

There are some pesky technicalities about typing contexts which need to be addressed. Althogh we have defined Γ as a *sequence* of variable-type mappings, the order shouldn't really be significant: $x : \texttt{Int}, y : \texttt{Int}$ is really the same thing as $y : \texttt{Int}, x : \texttt{Int}$. We essentially want to treat it as a set.

Formally, we can specify their equivalence by giving structural rules which say that a judgement holds in Γ if it holds in any permutation of Γ. Another convention is that any judgement which holds in a context Γ should hold in any bigger context Γ', where $\Gamma \subseteq \Gamma'$. For example, $x : \texttt{Int} \vdash x : \texttt{Int}$, but it is also true that $x : \texttt{Int}, y : \texttt{Int} \vdash x : \texttt{Int}$. In practice, the notation for contexts and the rules for how to manipulate them are so conventional that, beyond the quick summary in Figure 2.3., we will not bother to mention them again.

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

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

Figure 2.7: Structural rules for typing contexts.

Though `EBL` has no subtyping, most interesting languages do. This judgement is written form $\tau_1 <: \tau_2$ and it means that expressions of τ_1 may be provided anywhere expressions of τ_2 are expected, and the program will still be well-typed. A useful principle in the design and understanding of subtyping rules is Liskov's substitution principle, which states that if $\tau_1 <: \tau_2$, then instances of τ_2 can be replaced with instances of τ_1

without changing the program's semantic properties [7]. Subtyping rules are not usually totally semantic-preserving, but we'll occasionally use this idea to justify why certain subtyping rules are sensible.

2.1.4 Soundness

Having defined a type system, we want to know it is *sound*: that if the type system says a program is well-typed, the program will not run into type errors during execution. Soundness is a guarantee that our typing judgements, as we intuitively understand them, are mathematically correct. The exact definition of soundness depends on the language under consideration, but is often split into two parts called progress and preservation.

Theorem 1 (Progress). *If $\vdash e : \tau$ and e is not a value, then $e \longrightarrow e'$.*

Progress states that any well-typed, non-value term can be reduced i.e. it will not get stuck due to type errors. It also says that the definition of a value, as a non-variable irreducible expression, is coincident with the definition of a value as a particular category of terms in the grammar.

Theorem 2 (Preservation). *If $\vdash e : \tau$ and $e \longrightarrow e'$ then $\vdash e' : \tau$.*

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

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 *small-step soundness*. 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. This yields the following result.

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

These theorems are proven by structural induction on the typing rule used $\Gamma \vdash e : \tau$ or on the reduction rule used $e \to e'$.

In order to prove certain cases of progress and preservation there are two common lemmas needed. The first is canonical forms, which outlines a set of observations that follow immediately by observing 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 $\Gamma \vdash [e'/x]e : \tau$. In EBL, this lemma is needed to show that the reduction step in E-LET2 preserves soundness. The other languages considered in this report will have similar reduction steps.

A precise formulation of these two lemmas for EBL is given below.

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

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

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

Proofs for these lemmas and theorems can be found in Appendix A.

To summarise, soundness is a property which says, generally, that if a type system says a program is well-typed, it will not encounter a runtime type-error. The corollary of this is also interesting to consider: if a program has no runtime type-error, will the type system accept it? This property is called *completeness*, and almost no (interesting) type-systems are complete. This means a type system may reject type-safe programs. However, soundness guarantees that a type system will *always* reject programs which are *not* type-safe. Consider Figure 2.7., which demonstrates a type-safe Java program rejected by Java's type-system: the body of `double` is type-safe, because the conditional will always execute `returnx` $+$ `x`. However, Java will reject this program.

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

Figure 2.8: A type-safe Java method which does not typecheck.

Throughout this report we will only be concerned with sound type systems, but it is important to recognise that these type systems are all *conservative* because they may reject type-safe programs. 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" [14, 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 can make the system too conservative and of less practical use. This is an important trade-off we discuss in motivating $\lambda^{\rightarrow}_{\pi,\varepsilon}$.

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 [3], 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 $\lambda^{\rightarrow}_{\pi,\varepsilon}$. This section gives us an opportunity to familiarise ourself with $\lambda^{\rightarrow}$ to help introduce $\lambda^{\rightarrow}_{\pi,\varepsilon}$.

$$
\begin{array}{rcll}
e & ::= & & \textit{exprs}: \\
 & | & x & \textit{variable} \\
 & | & e\ e & \textit{application} \\
 & | & v & \textit{value} \\
 & & & \\
v & ::= & & \textit{values}: \\
 & | & \lambda x:\tau.e & \textit{abstraction}
\end{array}
\qquad
\begin{array}{rcll}
\tau & ::= & & \textit{types}: \\
 & | & B & \textit{base type} \\
 & | & \tau \rightarrow \tau & \textit{arrow type} \\
 & & & \\
\Gamma & ::= & & \textit{contexts}: \\
 & | & \varnothing & \textit{empty ctx.} \\
 & | & \Gamma, x:\tau & \textit{var. binding}
\end{array}
$$

Figure 2.9: Grammar for $\lambda^{\rightarrow}$.

Types in $\lambda^{\rightarrow}$ are either drawn from a set of base types B, or constructed using $\rightarrow$ ("arrow"). Given types τ_1 and τ_2, $\rightarrow$ can be used to compose a new type, $\tau_1 \rightarrow \tau_2$, 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` $\rightarrow$ `Bool`, `Bool` $\rightarrow$ `Int`, `Bool` $\rightarrow$ (`Bool` $\rightarrow$ `Int`). Arrow is right-associative, so `Bool` $\rightarrow$ `Bool` $\rightarrow$ `Int` = `Bool` $\rightarrow$ (`Bool` $\rightarrow$ `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.

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

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (\text{T-Var}) \qquad \frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x:\tau_1.e : \tau_1 \rightarrow \tau_2}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_3 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_3}\ (\text{T-App}) \qquad \frac{\Gamma \vdash e:\tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e:\tau_2}\ \text{T-Subsume}$$

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

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

Figure 2.10: Static rules for $\lambda^{\rightarrow}$.

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 $\tau_2 \to \tau_3$) 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 2.9.

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

Figure 2.11: 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 $\lambda^{\to}_{\pi,\varepsilon}$).

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

$[v/y]x = v$, 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 2.12: Substitution for $\lambda^{\to}$.

Substitution in $\lambda^{\to}$ 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 $\lambda^{\to}$. Such an expression is reduced by first reducing the left subexpression (E-APP1). For a well-typed expression, this will 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 $\lambda^{\to}$ are identical to those in `EBL`.

$\boxed{e \longrightarrow e}$

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

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

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

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 [5]. 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 $\lambda^{\rightarrow}_{\pi,\varepsilon}$, and not the expressive power of its dynamic semantics. Therefore we acknowledge this practical short-coming, but leave the basis of $\lambda^{\rightarrow}_{\pi,\varepsilon}$ 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 $\lambda^{\rightarrow}_{\pi,\varepsilon}$

2.3 Effect Systems

In the previous section we looked at how the static rules of a language might make a judgement such as $\Gamma \vdash e : \tau$, which ascribes the type τ to program e. This expresses a certain about what the runtime behaviour of the program is: namely that successive reductions of e will produce terms of type τ, and that this sequence of reductions will never get stuck due to a runtime type-error.

One extension to classical type systems is to incorporate a theory of *effects*. A *type-and-effect* system can ascribe a type and an effect to a piece of code, the effect component of which specifies intensional information about what will happen during the execution of the program [12]. For example, a judgement like $\Gamma \vdash e : \tau$ `with` $\{$`File.write`$\}$ means that successive reductions of e will result in terms of type τ, and during execution might write to a file. This tells us extra information about what can happen during runtime.

Talk about use of effect systems. Mention the basic effect system we will introduce.

2.3.1 ETL: Effect-Typed Language

2.4 The Capability Model

A *capability* is a unique, unforgeable reference, giving its bearer permission to perform some operation [4]. A piece of code S has *authority* over a capability C if it can directly invoke the operations endowed 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 a capability model, authority can only proliferate in the following ways [11]:

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 rules of authority proliferation are summarised as: "only connectivity begets connectivity".

Primitive capabilities are called *resources*. Resources model those initial capabilities passed into the runtime from the system environment. A capability is either a resource, or a function or object with (potentially transitive) authority over a capability. An example of a resource might be a particular file. A function which manipulates that file (for example, a logger) would also be a capability, but not a resource. Any piece of code which uses a capability, directly or indirectly, is called *impure*. For example, λ`x : Int. x` is pure, while λ`f : File. f.log("error message")` is impure.

A relevant concept in the design of capability-based programming languages is *ambient authority*. This is a kind of exercise of authority over a capability C which has not been explicitly [10]. Figure 2.4. gives an example in Java, where a malicious implementation of `List.add` attempts to overwrite the user's `.bashrc` file. `MyList` gains this capability by importing the `java.io.File` class, but its use of files is not immediate from the signature of its functions.

Ambient authority is 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.

```
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 2.14: `Main` exercises ambient authority over a `File` capability.

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

2.5 First-Class Modules

The exact way in which modules work is language-dependent, but we are particularly interested in languages with a first-class module systems. First-class modules are important in capability-safe languages because they mean capability-safe reasoning operates across module boundaries. Because modules are first-class, they must be instantiated like regular objects. They must therefore select their capabilities, and be supplied those capabilities by the proliferation rules of the capability model. In practice, first-class modules can be achieved by having module declarations desugar into an underlying lambda or object representation. This generally requires an "intermediate representation" of the language, which is simpler than the one in which programmers write.

Java is an example of a mainstream language whose modules are not first-class. Scala has first-class modules [13], 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 [6].

Chapter 3

Effect Calculi

3.1 $\lambda^{\rightarrow}_{\pi}$: Operation Calculus

The operation calculus $\lambda^{\rightarrow}_{\pi}$ is an extension of $\lambda^{\rightarrow}$ with primitive capabilities (resources) and their operations. Effects are identified with operation-calls. A program's runtime effects are those operations which it calls during execution. The static rules approximate the runtime effects of an expression, forming a simple effect system. These rules are very simple, but formalising and studying them will introduce new notations and a new concept of effect-soundness on which we shall build the $\lambda^{\rightarrow}_{\pi,\varepsilon}$.

3.1.1 Definition of $\lambda^{\rightarrow}_{\pi}$

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

Figure 3.1: Grammar for $\lambda^{\rightarrow}_{\pi}$.

The base types of $\lambda^{\rightarrow}_{\pi}$ are sets of resources, denoted by $\{\bar{r}\}$. Resources are drawn from a fixed set R of variables, and model those initial capabilities passed in from the system environment. They cannot be created at runtime. When a resource type is ascribed to a program, as in the judgement $\Gamma \vdash e : \{\bar{r}\}$, it means that if e terminates it will result in a resource literal $r \in \bar{r}$. A resource literal r is a variable ranging over the elements of R.

An operation is a special action that can be invoked on a resource-typed expression. For example, we might invoke the `open` operation on a `File` resource. Operations are drawn from a fixed set Π of variables and cannot be created at runtime.

An effect is an operation performed on a resource. Formally, they are members of $R \times \Pi$, but for readability we write `File.write` over $(\texttt{File}, \texttt{write})$. A set of effects is denoted by ε. Effects and operations notationally look the same, but should be distinguished: an effect is some action upon a resource which may happen during runtime; an operation-call is an expression representing the actual invocation of an effect at runtime.

In practical applications, operations take arguments. For example, when writing to a file, we want to specify *what* is being written to the file, e.g. `File.write("mymsg")`. Because we are only concerned with the use and propagation of effects, and not the semantics of particular effects, we make the simplifying assumption that all operations are null-ary.

The only type constructor is $\rightarrow_\varepsilon$, where ε is a concrete set of effects. $\tau_1 \rightarrow_\varepsilon \tau_2$ is the type of a function which takes inputs of type τ_1, produces outputs of type τ_2, and incurs no more effects than those contained in ε. For example, the type of a function which sends a message over a socket and returns a success flag could be $\texttt{Str} \rightarrow_{\texttt{Socket.write}} \texttt{Bool}$. From this signature we can tell this function will not open or close the socket, because the annotation on the arrow does not have those effects. A valid implementation of this function might not write to the `Socket`, because $\{\texttt{Socket.write}\}$ is an upper-bound on the effects which can happen. Every function type in $\lambda^{\rightarrow}_{\pi}$ must be annotated with an upper-bound in this way.

The static rules for $\lambda^{\rightarrow}_{\pi}$ are summarised in Figure 3.2. The typing judgement $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$ means that successive reductions on e will yield terms of type τ, and collectively incur no more than those effects in ε. This ε is a conservative approximation to the runtime effects of executing e, so it may contain effects which don't happen at runtime.

The rules for variables and values are: ε-VAR, ε-RESOURCE, and ε-ABS. These are identical to the rules in $\lambda^{\rightarrow}$, except they approximate the set of effects as $\varnothing$. Although a fucntion and a resource literal both encapsulate capabilities, something must be done to them (apply the function, operate on the resource) to incur a runtime effect.

The effects of a lambda application are: the effects of evaluating its subexpressions, and the effects incurred by executing the body of the lambda to which the left-hand side evaluates. Those last effects are obtained from the label on the lambda's arrow-type in the first premise.

The effects of an operation call are: the effects of evaluating the subexpression, and the single effect incurred when the subexpression is reduced to a resource literal r, and operation π is invoked on it. It is not always possible to know statically which exact resource literal the subexpression reduces to (if it halts at all). For example, the program `(if System.randomBool then File else Socket).close` may either reduce to `File.close`

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

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

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

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

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

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

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

Figure 3.2: Type-with-effect judgements in $\lambda^{\rightarrow}_{\pi}$.

or `Socket.close`. In such cases, the safe approximation is to type the conditional as $\{\texttt{File}, \texttt{Socket}\}$. ε-OPERCALL would then approximate the runtime effects of the operation call as $\{\texttt{File.close}, \texttt{Socket.close}\}$. Being able to type an expression as a (non-singleton) set of resources requires an extra rule: S-RESOURCE. This says that a subset of resources is also a subtype.

Because we're not really interested in what exactly an operation call does, every operation is valid for every resource. This can give bizarre programs — `Sensor.readTemp` seems like a sensible operation call, but what about `File.readTemp`? Because we are not interested in the specific behaviours of operations, we permit any operation on any resource to simplify the static rules.

The other subtyping rule is S-ARROW, a modification of the rule from $\lambda^{\rightarrow}$. In addition to this rule being contravariant in the input and covariant in the output, it is also covariant in the effects. Justified in terms of the Liskov substitution principle, any possible effect which might be incurred by the subtype should be expected by the supertype, otherwise substitution of a supertype for a subtype would allow for the possibility of new effects not possible under the original.

Figure 3.2. shows the updated definition of `substitution`. In $\lambda^{\rightarrow}_{\pi}$, a variable may only be substituted for a value, making the function a partial one. This restriction is imposed because, if a variable can be replaced with an arbitrary expression, then we might also be introducing arbitrary effects — a situation which violates the preservation of effects under reduction.

`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.

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

$$\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1'\, e_2 \mid \varepsilon}\ \text{(E-APP1)} \qquad \frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1\, e_2 \longrightarrow v_1\, e_2' \mid \varepsilon}\ \text{(E-APP2)} \qquad \frac{}{(\lambda x : \hat{\tau}.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x]\hat{e} \mid \varnothing}\ \text{(E-APP3)}$$

$$\frac{\hat{e} \rightarrow \hat{e}' \mid \varepsilon}{\hat{e}.\pi \longrightarrow \hat{e}'.\pi \mid \varepsilon}\ \text{(E-OPERCALL1)} \qquad \frac{r \in R \quad \pi \in \Pi}{r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}}\ \text{(E-OPERCALL2)}$$

Figure 3.4: Single-step reductions.

Single-step reduction is now a relation on an expression e and a pair $e \times \varepsilon$, representing the reduced expression and all effects incurred during the single-step of computation. E-APP1 and E-APP2 incur whatever is the effect of reducing their subexpressions. E-APP3 incurs no effects.

The new single-step rules are E-OPERCALL1 and E-OPERCALL2. The former reduces the receiver of an operation-call, and the latter performs an operation on a resource literal. E-OPERCALL1 incurs whatever is the effect of reducing the subexpression. E-OPERCALL2, which reduces the operation-call $r.\pi$, incurs the effect $r.\pi$.

Operation calls reduce to `unit` (which is a derived form; see Encodings). `unit` is a value representing the absence of information (because it is the only value of its type). Because we are not interested in the semantics of effects, we choose `unit` as the result of reducing an operation-call.

A multi-step reduction consists of zero[1] or more single-step reductions. The resulting effect-set is the union of all the single-steps taken.

[1]We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation[14, p. 39].

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

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

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

Figure 3.5: Multi-step reductions in $\lambda_{\pi}^{\rightarrow}$.

3.1.2 Soundness of $\lambda_{\pi}^{\rightarrow}$

Our goal is to show $\lambda_{\pi}^{\rightarrow}$ is sound. This requires an appropriate notion of *effect-soudnness*, as we need both the type and approximated effects of an expression to be preserved under reduction. Intuitively, a reduction $e_A \longrightarrow^{*} e_B \mid \varepsilon$ is sound if the type system's approximation of the effects of e contains the actual runtime effects ε, because then it has statically accounted for every possible runtime effect. Below is a definition, phrased in terms of single-step reduction.

Theorem 5 (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* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

This definition is slightly stronger, because it relates the approximation after reduction to the approximation before reduction, by saying it can only get more precise. This is analogous to the type-safe component of the definition, which states that the types of successive terms under reduction can only get more precise.

Our road to proving soundness takes the standard approach of showing that progress and preservation hold of $\lambda_{\pi}^{\rightarrow}$. Progress follows immediately by observing some properties of the typing rules.

Lemma 3 (Canonical Forms). *The following are true:*

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

Theorem 6 (Progress). *If* $\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon$ *and* $\hat{e}$ *is not a value, then* $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon$.

Proof. By induction on $\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon$, for $\hat{e}$ not a value. If the rule is ε-Subsumption it follows by inductive hypothesis. If $\hat{e}$ 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 is preserved by the substitution in the rule E-App3. The formulation of the substitution lemam is largely the same as it is in $\lambda^{\rightarrow}$, except the expression being substituted must be a value. This strengthening of the lemma is not a problem as the dynamic rules employ a strict evaluation strategy, so any expressions are reduced to values before they are substituted.

Lemma 4 (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 $\Gamma, x : \tau' \vdash e : \tau \texttt{ with } \varepsilon$. □

With this lemma, we are ready to prove the preservation theorem.

Theorem 7 (Preservation). *If* $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ *and* $e_A \longrightarrow e_B \mid \varepsilon$, *then* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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

Case: ε-APP Then $e_A = e_1\ e_2$ and $e_1 : \tau_2 \rightarrow_\varepsilon \tau_3 \texttt{ with } \varepsilon_1$ and $\Gamma \vdash e_2 : \tau_2 \texttt{ with } \varepsilon_2$. 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 $\Gamma, x : \tau_2 \vdash e : \tau_3 \texttt{ with } \varepsilon_3$. 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 $\Gamma \vdash e_1 : \{\bar{r}\} \texttt{ with } \varepsilon_1$. 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 $v_1.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}$. By canonical forms, $\Gamma \vdash v_1 : \texttt{unit with } \{r.\pi\}$. Also, $\Gamma \vdash \texttt{unit} : \texttt{Unit with } \varnothing$. Then $\tau_B = \tau_A$. Also, $\varepsilon_C \cup \varepsilon_B = \{r.\pi\} = \varepsilon_A$.

□

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

Theorem 8 (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 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 follows by inductively applying single-step soundness on the length of a multi-step reduction.

Theorem 9 (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. If the length is 0 then $e_A = e_B$ and the result holds vacuously. If the length is 1 the result holds by soundness of single-step reductions. 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 $\lambda^{\rightarrow}_{\pi,\varepsilon}$: Epsilon Calculus

The effect calculus is based on the simply-typed lambda calculus $\lambda^{\rightarrow}$. There is one type constructor, $\rightarrow$. The base types are sets of resources, denoted by $\{\bar{r}\}$. Although the calculus has no primitive notions of integers or booleans, we shall assume these may be encoded as they are in the usual way (e.g. as Church numerals) and make free use of them in examples as though they were standard, for the sake of readability.

3.2.1 Definition of $\lambda^{\rightarrow}_{\pi,\varepsilon}$

Resources are drawn from a fixed set R of variables, and model those initial capabilities passed in from the system environment. Resources cannot be created at runtime. When a resource type is ascribed to a program, as in the judgement $\Gamma \vdash e : \{\bar{r}\}$, it means that if e terminates it will result in a resource literal $r \in \bar{r}$.

A value v is either a resource literal r or a lambda abstraction $\lambda x : \tau.e$. The other forms of an expression are lambda application $e\ e$, variable x, and operation $e.\pi$. An operation is an action invoked on a resource. For example, we might invoke the `open` operation on a `File` resource. Operations are drawn from a fixed-set Π of variables. They cannot be created at runtime.

An effect is an operation performed on a resource. Formally, they are members of $R \times \Pi$, but for readability we write `File.write` over (`File`, `write`). A set of effects is denoted by ε. Effects and operations notationally look the same, but should be distinguished: an effect is some action upon a resource which may happen during runtime; an operation is the actual invocation of an effect at runtime.

In a practical language, operations should take arguments. For example, when writing to a file, we want to specify *what* is being written to the file, ala `File.write`(“`mymsg`”). Because $\lambda^{\rightarrow}_{\pi,\varepsilon}$ is only concerned with the use and propagation of effects, and not the semantics of particular effects, we make the simplifying assumption that all operations are null-ary.

Expressions may be labelled with the set of effects they might incur during execution. This is achieved by annotating all arrow types inside the expression. If a metavariable represents a labelled expression, it will be written with a hat; if it represents an unlabelled expression, it will have no hat. Compare e and $\hat{e}$.

$$
\begin{array}{lllr}
e & ::= & & \mathit{exprs}: \\
 & | & x & \mathit{variable} \\
 & | & v & \mathit{value} \\
 & | & e\ e & \mathit{application} \\
 & | & e.\pi & \mathit{operation} \\
\\
v & ::= & & \mathit{values}: \\
 & | & r & \mathit{resource\ literal} \\
 & | & \lambda x : \tau . e & \mathit{abstraction} \\
\\
\hat{e} & ::= & & \mathit{labelled\ exprs}: \\
 & | & x & \\
 & | & r & \\
 & | & \lambda x : \hat{\tau} . \hat{e} & \\
 & | & \hat{e}\ \hat{e} & \\
 & | & \hat{e}.\pi & \\
 & | & \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e & \mathit{import} \\
\\
\hat{v} & ::= & & \mathit{labelled\ values}: \\
 & | & r & \\
 & | & \lambda x : \hat{\tau} . \hat{e} &
\end{array}
$$

$$
\begin{array}{lllr}
\varepsilon & ::= & & \mathit{effects}: \\
 & | & \{\overline{r.\pi}\} & \\
\\
\tau & ::= & & \mathit{types}: \\
 & | & \{\bar{r}\} & \\
 & | & \tau \to \tau & \\
\\
\hat{\tau} & ::= & & \mathit{labelled\ types}: \\
 & | & \{\bar{r}\} & \\
 & | & \hat{\tau} \to_{\varepsilon} \hat{\tau} & \\
\\
\Gamma & ::= & & \mathit{type\ ctx}: \\
 & | & \varnothing & \\
 & | & \Gamma, x : \tau & \\
\\
\hat{\Gamma} & ::= & & \mathit{labelled\ type\ ctx}: \\
 & | & \varnothing & \\
 & | & \hat{\Gamma}, x : \hat{\tau} &
\end{array}
$$

Figure 3.6: Effect calculus.

Labelling of an expression is *deep*. That is, every subterm of a labelled term is also labelled. Unlabelled terms are also deeply unlabelled. The only exception is the `import` expression, which is the only way to compose labelled and unlabelled code. `import` nests unlabelled code inside labelled code, and selects those capabilities ε over which the unlabelled code has authority. It is not possible to nest labelled code inside unlabelled code.

The distinction between labelled and unlabelled types and expressions requires us to have the notion of labelled and unlabelled contexts. Labelled contexts only bind variables to labelled types, whereas unlabelled contexts only bind variables to unlabelled types. There is no valid context which mixes labelled and unlabelled types.

A construct's labelled version is always denoted with a hat.

Given a piece of unlabelled code e and static effects ε we can produce a labelled piece of code $\texttt{annot}(e, \varepsilon) = \hat{e}$ by annotating every function with ε. In the reverse direction,

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

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

$\mathtt{annot} :: \boldsymbol{\tau} \times \boldsymbol{\varepsilon} \rightarrow \hat{\boldsymbol{\tau}}$

$$\begin{aligned}
&\mathtt{annot}(\{\bar{r}\}, \_) = \{\bar{r}\}\\
&\mathtt{annot}(\tau \rightarrow \tau, \varepsilon) = \tau \rightarrow_{\varepsilon} \tau.
\end{aligned}$$

$\mathtt{annot} :: \Gamma \times \boldsymbol{\varepsilon} \rightarrow \hat{\Gamma}$

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

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

$$\begin{aligned}
&\mathtt{erase}(\{\bar{r}\})\\
&\mathtt{erase}(\hat{\tau}_1 \rightarrow_{\varepsilon} \hat{\tau}_2) = \mathtt{erase}(\hat{\tau}_1) \rightarrow \mathtt{erase}(\hat{\tau}_2)
\end{aligned}$$

$\mathtt{erase} :: \hat{\mathtt{e}} \rightarrow \mathtt{e}$

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

Figure 3.7: Annotation functions.

given some labelled code $\hat{e}$ we can produce an unlabelled piece of code $\mathtt{erase}(\hat{e}) = e$ by removing the labels on functions. Full definitions for these functions on expressions, types, and contexts are given in Figure 3.2. Note that `erase` is undefined on `import` expressions. We won't ever need to erase import expressions, but it means the function is partial, so we need to be careful when we use it.

Annotation is not always safe. For instance, $\mathtt{annot}(\lambda l : \mathtt{Int} \rightarrow_{File.read} \mathtt{Int}.\ l\ 1, \varnothing)$ would overwrite the `File.read` effect permitted by l. `annot` is used in one place, in the dynamic rules, and for that limited use we will have to prove its safety.

We may wish to know what effects are encapsulated by a piece of labelled code. This is achieved by two functions, $\mathtt{effects}(\hat{e})$ and $\mathtt{ho\text{-}effects}(\hat{e})$, which collectively compute the set of effects captured by $\hat{e}$. These are effects which may, directly or indirectly, be invoked by $\hat{e}$. The difference between the two functions is in who supplies the effect. $\mathtt{effect}(\hat{e})$ is the set of effects for which $\hat{e}$ has direct authority, while `ho-effects` is the

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

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

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

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

Figure 3.8: Effect functions.

set of effects for which $\hat{e}$ has (strictly) transitive authority. These higher-order effects are always supplied by some external environment.

For example, take the function which, given a file, reads and returns its contents (which are perhaps encoded as an integer). Its signature would be $\texttt{f} : \{\texttt{File}\} \rightarrow_{\texttt{File.read}} \texttt{Int}$. The $\texttt{effects}(\texttt{f}) = \{\texttt{File.read}\} \cup \texttt{effects}(\texttt{Int})$, because any client using `f` will directly invoke the `File.read` operation and may use any resource encapsulated by the `Int` type. The $\texttt{ho-effects}(\texttt{f}) = \{\texttt{File}.\pi \mid \pi \in \Pi\}$, because to use `f` it must be supplied with a `File` literal from some outside source. Therefore, every possible effect on `File` is a higher-order effect.

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

$[\hat{v}/y]x = \hat{v}$, 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 3.9: Substitution function.

The substitution function $\texttt{substitution}(\hat{e}, \hat{v}, x)$ replaces all free occurrences of x with $\hat{v}$ in $\hat{e}$. The short-hand is $[\hat{v}/x]\hat{e}$. When performing multiple substitutions we use the notation $[\hat{v}_1/x_1, \hat{v}_2/x_2]\hat{e}$ as shorthand for $[\hat{v}_2/x_2]([\hat{v}_1/x_1]\hat{e})$. Note how the order of the variables has been flipped; the substitutions occur as they are written, left-to-right.

Note that substitution is partial, because it is only defined when a free-variable is being replaced with a value. This is important for proving preservation, because if we replace variables with arbitrary expressions, then we might also be introducing arbitrary effects.

To avoid accidental variable capture we adopt the convention of α-conversion, whereby we freely and implicitly interchange expressions which are equivalent up to the naming of bound variables [14, p. 71]. This elides some tedious bookkeeping. Consequently, we shall assume variables are (re-)named in this way to avoid accidental capture.

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\textsc{T-Var}) \qquad \frac{}{\Gamma, r : \{r\} \vdash r : \{r\}}\ (\textsc{T-Resource}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ (\textsc{T-Abs})$$

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

Figure 3.10: Typing judgements in the epsilon calculus.

The first sort of static judgement ascribes a type to a piece of unlabelled code. T-VAR, T-APP, and T-OPERCALL are the same as they are in $\lambda^{\rightarrow}$. T-RESOURCE is the same as T-VAR, but for variables representing primitive capabilities. T-OPERCALL is the rule for typing an operation call $e_1.\pi$. Such an expression is well-typed if e_1 types to some valid resource, and π is a known operation.

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

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

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

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

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

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

Figure 3.11: Safety judgements in the epsilon calculus.

Before presenting the type-with-effect rules for labelled expressions, we first define a few safety predicates. Intuitively, the type $\hat{\tau}$ is `safe` for ε if it has declared every (non higher-order) effect $r.\pi \in \varepsilon$ in its signature. $\hat{\tau}$ is `ho-safe` for ε if $\hat{\tau}$ has declared every higher-order effect $r.\pi \in \varepsilon$ in its signature. One way to think about these predicates is as a contract between caller and callee. If the caller supplies a set of capabilities ε to a piece of code typing to $\hat{\tau}$, it would violate the restriction on *ambient authority* if a capability was supplied that $\hat{\tau}$ had not explicitly asked for. Therefore, $\texttt{safe}(\hat{\tau}, \varepsilon)$ holds when the (non higher-order) effects selected by $\hat{\tau}$ include ε. $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ holds when the higher-order effects selected by $\hat{\tau}$ include ε.

Because the implementation of $\hat{\tau}$ might internally propagate capabilities, the definitions of safety and higher-order safety need to be transitive. **Give an example of why this is so.**

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

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

$$\frac{\hat{\Gamma}, x : \hat{\tau}_2 \vdash \hat{e} : \hat{\tau}_3 \texttt{ with } \varepsilon_3}{\hat{\Gamma} \vdash \lambda x : \tau_2.\hat{e} : \hat{\tau}_2 \to_{\varepsilon_3} \hat{\tau}_3 \texttt{ with } \varnothing} \; (\varepsilon\text{-ABS}) \qquad \frac{\hat{\Gamma} \vdash \hat{e}_1 : \hat{\tau}_2 \to_{\varepsilon} \hat{\tau}_3 \texttt{ with } \varepsilon_1 \quad \hat{\Gamma} \vdash \hat{e}_2 : \hat{\tau}_2 \texttt{ with } \varepsilon_2}{\hat{\Gamma} \vdash \hat{e}_1\hat{e}_2 : \hat{\tau}_3 \texttt{ with } \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon} \; (\varepsilon\text{-APP})$$

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

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

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

Figure 3.12: Type-with-effect judgements.

$\lambda^{\to}_{\pi,\varepsilon}$ has a new kind of judgement. $\Gamma \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon$ can be read as saying that $\hat{e}$, if it halts, will produce a value of type $\hat{\tau}$ and incur at most the set of effects ε. This judgement gives a conservative approximation as to what will happen; some of the effects in the typing judgement may not actually happen at runtime.

The simplest rules are those which type values as having no effect. Although a function and a resource literal can both capture capabilities, you must do something with them (apply the function, operate on the resource) to incur a runtime effect.

The effects of a lambda application are: the effects of evaluating its subexpressions, and the effects incurred by executing the body of the lambda to which the left-hand side evaluates. Those last effects are pulled from the label on the lambda's arrow-type.

The effects of an operation call are: the effects of evaluating the subexpression, and the single effect incurred when the subexpression is reduced to a resource literal r, and operation π is invoked on it. It is not always possible to know statically which exact resource literal the subexpression reduces to (if it halts at all). Figure 3.8. shows such an example. The safe approximation is to say that the operation call $\hat{e}.\pi$ incurs π on every possible resource to which $\hat{e}$ might evaluate. In the case of Figure 3.8., this would be $\{\texttt{File.write}, \texttt{Socket.write}\}$.

Because we're not really interested in what exactly an operation call does, every operation is valid for every resource. This can give bizarre programs — `Sensor.readTemp` seems like a sensible operation call, but what about `File.readTemp`? — however, an adequate treatment is outside of the scope of $\lambda^{\rightarrow}_{\pi,\varepsilon}$, so we gloss over such programs.

It actually might be possible to figure out the exact literal if the system's not Turing complete, since the simply-typed lambda calculus is strongly normalising (and this is basically that, with a few extras), so be careful about this claim

```
def getResource(b: Bool): { File, Socket } with ∅ =
   if b then File else Socket

val boolVal: Bool = System.randomBool
getResource(boolVal).write
```

Figure 3.13: We cannot statically determine which branch will execute, so the safe approximation for `getResource(boolVal).write` is $\{\texttt{File.write}, \texttt{Socket.write}\}$.

The most interesting rule is ε-Import. This rule is set up to ensure the interaction between labelled and unlabelled code is capability-safe. We type e with $x : \texttt{erase}(\hat{\tau})$. This eliminates ambient authority, because the only free variables in e will be those selected by the interface $\hat{\tau}$.

For our rule to be capability-safe, we need to ensure that any higher-order function in scope is expecting the set of capabilities in $\hat{\tau}$. If not, we could exercise ambient authority by passing that higher-order function a capability from $\hat{\tau}$ which it hadn't selected. This is the purpose of $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$: all higher-order functions in scope need to be expecting any capability they might be passed.

In the conclusion of the rule we annotate the unlabelled code's effects as $\texttt{effects}(\hat{\tau})$. Because this is the full set of capabilities over which e has access, and because this set is higher-order safe, we shall see this annotation is sound.

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

$$\frac{\varepsilon \subseteq \varepsilon' \quad \hat{\tau}_2 <: \hat{\tau}_2' \quad \hat{\tau}_1' <: \hat{\tau}_1}{\hat{\tau}_1 \rightarrow_{\varepsilon} \hat{\tau}_2 <: \hat{\tau}_1' \rightarrow_{\varepsilon'} \hat{\tau}_2'} \ (\text{S-EFFECTS}) \qquad \frac{r \in r_1 \implies r \in r_2}{\{\bar{r}_1\} <: \{\bar{r}_2\}} \ (\text{S-RESOURCES})$$

Figure 3.14: Subtyping judgements in the epsilon calculus.

In addition to the usual subtyping rules from $\lambda^{\rightarrow}$ between τ terms, we introduce two more for $\hat{\tau}$ terms.

The rule for functions is contravariant in the input-type and covariant in the output-type (as in $\lambda^{\rightarrow}$), and requires the effects of the super-type to be an upper-bound of the effects of the sub-type. We can think of this in terms of Liskov's substitution principle: if the subtype incurred an effect the supertype hadn't declared, it would violate the supertype's interface.

The rule for resources says that a superset of resources is a subtype.

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

$$\frac{\hat{e}_1 \longrightarrow \hat{e}_1' \mid \varepsilon}{\hat{e}_1\hat{e}_2 \longrightarrow \hat{e}_1'\hat{e}_2 \mid \varepsilon}\ (\textsc{E-App1}) \qquad \frac{\hat{e}_2 \longrightarrow \hat{e}_2' \mid \varepsilon}{\hat{v}_1\hat{e}_2 \longrightarrow \hat{v}_1\hat{e}_2' \mid \varepsilon}\ (\textsc{E-App2}) \qquad \frac{}{(\lambda x : \hat{\tau}.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x]\hat{e} \mid \varnothing}\ (\textsc{E-App3})$$

$$\frac{\hat{e} \rightarrow \hat{e}' \mid \varepsilon}{\hat{e}.\pi \longrightarrow \hat{e}'.\pi \mid \varepsilon}\ (\textsc{E-OperCall1}) \qquad \frac{r \in R \quad \pi \in \Pi}{r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}}\ (\textsc{E-OperCall2})$$

$$\frac{\hat{e} \longrightarrow \hat{e}' \mid \varepsilon'}{\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e \longrightarrow \texttt{import}(\varepsilon)\ x = \hat{e}'\ \texttt{in}\ e \mid \varepsilon'}\ (\textsc{E-Import1})$$

$$\frac{}{\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon) \mid \varnothing}\ (\textsc{E-Import2})$$

Figure 3.15: Single-step reductions.

A single-step reduction takes an expression to a pair consisting of an expression and a set of runtime effects. The rules E-APP1, E-APP2, E-OPERCALL1, E-IMPORT1 all reduce a single subexpression.

E-APP3 is the standard $\lambda^{\rightarrow}$ rule for applying a value to a lambda, by performing substitution on the lambda body.

E-OPERCALL2 performs an operation on a resource literal. In this case it reduces to `unit` (which is a derived form in our calculus; see 3.4. Encodings). This choice reflects the fact that $\lambda^{\rightarrow}_{\pi,\varepsilon}$ doesn't model the potentially varied return types of functions.

E-IMPORT2 performs module resolution. The (unlabelled) body of code is annotated with the set of effects captured by the interface, and then the value being imported is substituted into the body of code.

A multi-step reduction consists of zero[2] or more single-step reductions. The resulting effect-set is the union of all the single-steps taken.

3.2.2 Soundness of $\lambda^{\rightarrow}_{\pi,\varepsilon}$

Our goal is to show $\lambda^{\rightarrow}_{\pi,\varepsilon}$ is sound. This requires an appropriate notion of *effect-soundness*.

[2]We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation[14, p. 39].

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

$$\frac{}{\hat{e} \to^* \hat{e} \mid \varnothing}\ (\text{E-MultiStep1}) \quad \frac{\hat{e} \to \hat{e}' \mid \varepsilon}{\hat{e} \to^* \hat{e}' \mid \varepsilon}\ (\text{E-MultiStep2})$$

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

Figure 3.16: Multi-step reductions.

Theorem 10 (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 $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.*

This definition of soundness is the same as in $\lambda^{\rightarrow}$ but for an extra conclusion: $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$. Intuitively, ε_A is the approximation of what runtime effects the reduction of $\hat{e}_A$ will incur, ε is the actual set of effects $\hat{e}_A$ incurred (at most a singleton because we are working with single-step reduction), and ε_B is the approximation of what runtime effects the reduction of $\hat{e}_B$ will incur. Evidently we want $\varepsilon \subseteq \varepsilon_A$; an approximation which accounts for every runtime effect is a sound one. We also want $\varepsilon_B \subseteq \varepsilon_A$, so successive approximations only get better.

The soundness proof takes the standard approach of showing that progress and preservation hold of the calculus. This can be done immediately by observing some properties that follow immediately from the typing rules.

Lemma 5 (Canonical Forms). *The following are true:*

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

Theorem 11 (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, for $\hat{e}$ not a value. If the rule is ε-Subsumption it follows by inductive hypothesis. If $\hat{e}$ has a reducible subexpression then reduce it. Otherwise use one of ε-App3, ε-OperCall2, or ε-Import2. □

To prove preservation, we need to know types and effects are preserved under substitution. The substitution lemma gives us this result. It says that if x is bound to a type, and a value $\hat{v}$ of that type is substituted into $\hat{e}$, then the type and effect of $\hat{e}$ remain unchanged. Key to this property is that $\hat{v}$ is a value, so by canonical forms it cannot introduce effects that weren't already in $\hat{e}$. Beyond this observation, the proof is routine.

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. □

The tricky case in preservation is when an `import` expression is resolved. To show the reduction $\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon) \mid \varnothing$ preserves soundness requires a few things. First, if $\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e : \hat{\tau}_A\ \texttt{with}\ \varepsilon_A$, then we need to be able to type the reduced expression in the same context: $\hat{\Gamma} \vdash [\hat{v}/x]\texttt{annot}(e, \varepsilon) : \hat{\tau}_B\ \texttt{with}\ \varepsilon_B$. To be effect-sound, we need $\varepsilon_B \subseteq \varepsilon_A$. To be type-sound, we need $\hat{\tau}_B <: \hat{\tau}_A$. This motivates the next lemma, which relates a typing judgement of e to a typing judgement of $\texttt{annot}(e, \varepsilon)$.

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

- $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}\ \texttt{with}\ \varnothing$
- $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e : \tau$
- $\varepsilon = \texttt{effects}(\hat{\tau})$
- $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$

Then $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$.

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

The exact formulation of the Annotation lemma is very specific to the premises of ε-IMPORT2, but generalised slightly to accommodate a proof by induction. The generalisation is to allow e to be typed in any context Γ with a binding for y. We can think of Γ as encapsulating the ambient authority exercised by e. At the top-level of any program, we will always have $\Gamma = \varnothing$, because the typing judgement ε-IMPORT always types `import` expressions with just the authority being selected. However, inductively-speaking, there may be ambient capabilities. Consider $(\lambda x : \{\texttt{File}\}.\ \texttt{x.write})\ \texttt{File}$. From the perspcetive of `x.write`, `File` is an ambient capability, and so if we were to inductively apply the Annotation lemma, at this point, $\texttt{File} \in \Gamma$. However, because the code encapsulating `x.write` selects `File` by binding it to x in the function declaration, this code is capability-safe.

Proving the annotation lemma requires an additional pair of lemmas, to relate $\hat{\tau}$ and $\texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$.

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`. □

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 $\hat{\tau} = \hat{\tau}_1 \rightarrow_{\varepsilon'} \tau_2$ 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 are now ready to prove the preservation theorem.

Theorem 12 (Preservation). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A *and* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$*, then* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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

Case: ε-APP Then $e_A = \hat{e}_1\hat{e}_2$ and $\hat{e}_1 : \hat{\tau}_2 \rightarrow_\varepsilon \hat{\tau}_3$ `with` ε_1 and $\hat{\Gamma} \vdash \hat{e}_2 : \hat{\tau}_2$ `with` ε_2. If the reduction rule used was E-APP1 or E-APP2, then the result follows by applying the inductive hypothesis to $\hat{e}_1$ and $\hat{e}_2$ respectively.

Otherwise the rule used was E-APP3. Then $(\lambda x : \hat{\tau}_2.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x]\hat{e} \mid \varnothing$. By inversion on the typing rule for $\lambda x : \hat{\tau}_2.\hat{e}$ we know $\Gamma, x : \hat{\tau}_2 \vdash \hat{e} : \hat{\tau}_3$ `with` ε_3. By canonical forms, $\varepsilon_2 = \varnothing$ because $\hat{e}_2 = \hat{v}_2$ is a value. Then by the substitution lemma, $\hat{\Gamma} \vdash [\hat{v}_2/x]\hat{e} : \hat{\tau}_3$ `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 $\hat{\Gamma} \vdash e_1 : \{\bar{r}\}$ `with` ε_1. If the reduction rule used was E-OPERCALL1 then the result follows by applying the inductive hypothesis to $\hat{e}_1$.

Otherwise the reduction rule used was E-OPERCALL2 and $v_1.\pi \longrightarrow$ `unit` $\mid \{r.\pi\}$. By canonical forms, $\hat{\Gamma} \vdash v_1$: `unit` `with` $\{r.\pi\}$. Also, $\hat{\Gamma} \vdash$ `unit` : `Unit` `with` $\varnothing$. Then $\tau_B = \tau_A$. Also, $\varepsilon_C \cup \varepsilon_B = \{r.\pi\} = \varepsilon_A$.

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

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

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

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

Our statement of soundness combines the progress and preservation theorems into one.

Theorem 13 (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* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

Proof. If $\hat{e}_A$ 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, multi-step reductions can straightforwardly be be shown to also be sound. This is done by inductively applying single-step soundness to the length of the multi-step reduction.

Theorem 14 (Multi-step Soundness). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ `with` ε_A *and* $e_A \longrightarrow^* e_B \mid \varepsilon$*, where* $\hat{\Gamma} \vdash e_B : \hat{\tau}_B$ `with` ε_B *and* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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 1 the result holds by soundness of single-step reductions. 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. □

Chapter 4

Applications

4.1 Encodings

When writing practical examples it is useful to use higher-level constructs which have been derived from the base language. In this section we introduce some of the constructs that we use in examples. Because the core language is sound, any derived extension is also sound.

4.1.1 Unit

`Unit` is a type inhabited by exactly one value. It conveys the absence of information. In our dynamic rules, `unit` is what an operation call on a resource literal is reduced to. We define `unit` $\overset{\text{def}}{=} \lambda x : \varnothing . x$ and `Unit` $\overset{\text{def}}{=} \varnothing \rightarrow_{\varnothing} \varnothing$. Note that because there is no empty resource literal, `unit` cannot be applied to anything. Furthermore, $\vdash$ `unit` : `Unit with` $\varnothing$, by ε-ABS, so any context can make this type judgement.

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

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

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

Figure 4.1: Derived `Unit` rules.

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` $\hat{e}_2 \overset{\text{def}}{=} (\lambda x : \hat{\tau}_1 . \hat{e}_2)\hat{e}_1$. Note that if $\hat{e}_1$ is a non-value, we can reduce the `let` by E-APP2. If $\hat{e}_1$ is a value, we may apply E-APP3, which binds $\hat{e}_1$ to x in $\hat{e}_2$. This is fundamentally a lambda application, so it can be typed using ε-APP (or T-APP, if the terms involved are unlabelled).

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

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

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

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

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

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

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

Figure 4.2: Derived `let` rules.

4.1.3 Conditionals

4.1.4 Tuples

We need tuples to import multiple names.

4.2 Examples

EXAMPLE OF A RESOURCE LEAKING AND BREAKING CONFINEMENT

EXAMPLE OF IMPORTING MULTIPLE CAPABILITIES, ONE GETS LEAKED AND PASSED SOMEWHERE IT HASN'T BEEN SELECTED

```
resource module Logger
require File

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

```
require File
instantiate Logger(File)

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

Figure 4.3: A `logger` client doesn't need to add effect labels. These can be inferred.

```
resource module Logger
require File

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

```
module Client

def action(l: Logger): Unit with File.append =
   l.log()
```

```
require File
instantiate Logger(File)

def main(): Unit with File.append =
   Client.action(Logger)
```

Figure 4.4: This won't type because of a mismatch between the client's effects and the logger's effects.

Chapter 5

Evaluation

5.1 Related Work

Fengyun Liu has approached the study of capability-based effect systems by developing a lambda calculus based around two type-constructors for building free and stoic functions [8]. Free functions may ambiently capture capabilities, but stoic functions may not; for a stoic function to have any effect, it must be explicitly given the capability for that effect. The resulting theory allows the type system to determine if a stoic function is pure or not by inspecting its parameters. If a function is known to be pure there are many optimisations that can be made (inlining, parallelisation). Liu's work is largely motivated by achieving such optimiastions for Scala compilers.

By contrast, our work is motivated by the propagation and use of capabilities, and how language-design features might inform software design. Unlike Liu's System F-Impure, $\lambda^{\rightarrow}_{\pi,\varepsilon}$ has no effect-polymorphism. However, our work has more fine-grained detail about those effects incurred by a particular function — while System F-Impure can conclusively determine if a stoic function is pure, determining what particular effects an impure function has is outside of the scope of Liu's work.

5.2 Future Work

A major limitation to practical adoption of $\lambda^{\rightarrow}_{\pi,\varepsilon}$ is that it is not Turing complete — it has no general recursion, nor recursive types. Extending $\lambda^{\rightarrow}_{\pi,\varepsilon}$ to include these features would bring it up to par with real programming languages.

Miller's formulation of the capability-model is in terms of objects, and all of the capability-safe languages to which this paper has referred are object-oriented. It is worth investigating how the bridge between $\lambda^{\rightarrow}_{\pi,\varepsilon}$ and existing capability-safe languages might be bridged by investigating different object encodings, and determining which langauge extensions are needed to enable these. By extension, these languages have first-class modules, so a version of $\lambda^{\rightarrow}_{\pi,\varepsilon}$ which can reason about objects would immediately yield module-level reasoning.

The biggest contribution that could be made to $\lambda^{\rightarrow}_{\pi,\varepsilon}$ would be to enrich it with a theory of polymorphic effects. As an example, consider $\lambda x : \texttt{Unit} \rightarrow_{\varepsilon} \texttt{Unit}.\ x\ \texttt{unit}$, where ε is free. Invoking this particular function would incur every effect in ε, but allowing general. Currently $\lambda^{\rightarrow}_{\pi,\varepsilon}$ has no way to define such functions which are parametrised by effect-sets. Deveoping an extension which can handle polymorphic effects would be a valuable contribution, and improve the stock of $\lambda^{\rightarrow}_{\pi,\varepsilon}$ as a practical type-and-effect system.

5.3 Conclusion

$\lambda^{\rightarrow}_{\pi,\varepsilon}$ is an extension to $\lambda^{\rightarrow}$ which allows for the import of capabilities into unlabelled code. This importing is done in a capability-safe manner, which prohibits the exercise of ambient authority. As a result, we can safely bound the set of possible effects in the unlabelled code by inspecting those capabilities passed into it via the `import` expression.

Talk about examples given, mention any extensions needed to allow for things such as multiple imports.

There are some important limitations to $\lambda^{\rightarrow}_{\pi,\varepsilon}$: it has no general recursion, and no recursive types; it is formulated in terms of the lambda calculus, whereas the capability model is stated in terms of objects; it has no way to express functions with polymorphic effects. These are all interesting avenues of future work that would enrich $\lambda^{\rightarrow}_{\pi,\varepsilon}$ and our collective understanding of the relation between effects and capabilities.

Appendix A

$\lambda_{\pi}^{\rightarrow}$ Proofs

Lemma 10 (Canonical Forms)**.** *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 15 (Progress)**.** *If* $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$ *and* e *is not a value, then* $e \longrightarrow e' \mid \varepsilon$.

Proof. By induction on $\Gamma \vdash e : \tau \texttt{ with } \varepsilon$, for e not a value.

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

Case: ε-OPER. Then $e = e_1.\pi$. If e_1 is a non-value, 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, $v_1 = r$ and $\Gamma \vdash v_1 : \{r\} \texttt{ with } \varnothing$. 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 $\varepsilon' \subseteq \varepsilon$. These are subderivations, so the result holds by inductive assumption.

$\square$

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 $y \neq x$. If $y \neq x$. Then $[v/x]y = y$ and $\Gamma \vdash y : \tau \texttt{ with } \varnothing$. Therefore $\Gamma \vdash [v/x]y : \tau \texttt{ with } \varnothing$. Otherwise $y = x$. By inversion on

ε-VAR, the typing judgement from the theorem assumption is $\Gamma, x : \tau' \vdash x : \tau'$ with $\varnothing$. Since $[v/x]y = v$, and by assumption $\Gamma \vdash v : \tau'$ with $\varnothing$, then $\Gamma \vdash [v/x]x : \tau'$ with $\varnothing$.

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

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

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

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

□

Theorem 16 (Preservation). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ with ε_A *and* $e_A \longrightarrow e_B \mid \varepsilon_C$*, then* $\hat{\Gamma} \vdash e_B : \tau_B$ with ε_B*, where* $e_B <: e_A$ *and* $\varepsilon \cup \varepsilon_B \subseteq \varepsilon_A$.

Proof. By induction on $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ with ε_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 $e_A = \hat{e}_1\,\hat{e}_2$ and $\hat{e}_1 : \hat{\tau}_2 \rightarrow_{\varepsilon} \hat{\tau}_3$ with ε_1 and $\hat{\Gamma} \vdash \hat{e}_2 : \hat{\tau}_2$ with ε_2.

Subcase: E-APP1. Todo.

Subcase: E-APP2. Todo.

Subcase: E-APP3. Then $(\lambda x : \hat{\tau}_2.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x]\hat{e} \mid \varnothing$. By inversion on the typing rule for $\lambda x : \hat{\tau}_2.\hat{e}$ we know $\Gamma, x : \hat{\tau}_2 \vdash \hat{e} : \hat{\tau}_3$ with ε_3. By canonical forms, $\varepsilon_2 = \varnothing$ because $\hat{e}_2 = \hat{v}_2$ is a value. Then by the substitution lemma, $\hat{\Gamma} \vdash [\hat{v}_2/x]\hat{e} : \hat{\tau}_3$ 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.

Subcase: E-OPERCALL1.

Subcase: Otherwise the reduction rule used was E-OPERCALL2 and $v_1.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}$. By canonical forms, $\hat{\Gamma} \vdash v_1 : \texttt{unit with } \{r.\pi\}$. Also, $\hat{\Gamma} \vdash \texttt{unit} : \texttt{Unit with } \varnothing$. Then $\tau_B = \tau_A$. Also, $\varepsilon_C \cup \varepsilon_B = \{r.\pi\} = \varepsilon_A$. □

Theorem 17 (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* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

Proof. If $\hat{e}_A$ is not a value then the reduction exists by the progress theorem. The rest follows by the preservation theorem. □

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

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 $\hat{e}_A \longrightarrow^* \hat{e}_C \mid \varepsilon'$ 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, $\hat{\Gamma} \vdash \hat{e}_C : \hat{\tau}_C \texttt{ with } \varepsilon_C$ and $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$. By inductive assumption, $\hat{\tau}_C <: \hat{\tau}_A$ and $\hat{\varepsilon}_C \cup \varepsilon' \subseteq \varepsilon_A$. By single-step soundness, $\hat{\tau}_B <: \hat{\tau}_C$ and $\hat{\varepsilon}_B \cup \varepsilon'' \subseteq \varepsilon_C$. Then by transitivity, $\hat{\tau}_B <: \hat{\tau}$ and $\hat{\varepsilon}_B \cup \varepsilon' \cup \varepsilon'' = \varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$. □

Appendix B

$\lambda^{\rightarrow}_{\pi,\varepsilon}$ Proofs

Lemma 12 (Canonical Forms). *The following are true:*

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

Theorem 19 (Progress). *If* $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon$ *and* $\hat{e}$ *is not a value, then* $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon$.

Proof. By induction on $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon$, for $\hat{e}$ not a value.

Case: ε-APP. Then $\hat{e} = \hat{e}_1\ \hat{e}_2$. If $\hat{e}_1$ is a non-value, then $\hat{e}_1\ \hat{e}_2 \longrightarrow \hat{e}_1'\ \hat{e}_2$ by E-APP1. If $\hat{e}_1 = \hat{v}_1$ is a value and $\hat{e}_2$ is a non-value, then $\hat{e}_1\ \hat{e}_2 \longrightarrow \hat{v}_1\ \hat{e}_2'$ by E-APP2. Otherwise $\hat{e}_1$ and $\hat{e}_2$ are both values. By inversion, $\hat{e}_1 = \lambda x : \hat{\tau}.\hat{e}$, so $(\lambda x : \hat{\tau}.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x] \mid \varnothing$ by E-APP3.

Case: ε-OPER. Then $\hat{e} = \hat{e}_1.\pi$. If $\hat{e}_1$ is a non-value, then $\hat{e}_1.\pi \longrightarrow \hat{e}_1'.\pi \mid \varepsilon_1$ by E-OPERCALL1. Otherwise $\hat{e}_1 = \hat{v}_1$ is a value. By canonical forms, $\hat{v}_1 = r$ and $\hat{\Gamma} \vdash v_1 : \{r\}\ \texttt{with}\ \varnothing$. Then $r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}$ by E-OPERCALL2.

Case: ε-SUBSUME. Then $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}'\ \texttt{with}\ \varepsilon'$. By inversion, $\hat{\Gamma} \vdash \hat{e} : \tau\ \texttt{with}\ \varepsilon$, where $\tau' <: \tau$ and $\varepsilon' \subseteq \varepsilon$. These are subderivations, so the result holds by inductive assumption.

Case: ε-MODULE. Then $\hat{e} = \texttt{import}(\varepsilon)\ x = \hat{e}'\ \texttt{in}\ e$. If $\hat{e}'$ is a non-value 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 13 (Substitution). *If* $\hat{\Gamma}, x : \hat{\tau}' \vdash e : \hat{\tau}\ \texttt{with}\ \varepsilon$ *and* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}'\ \texttt{with}\ \varnothing$ *then* $\hat{\Gamma} \vdash [\hat{v}/x]e : \hat{\tau}\ \texttt{with}\ \varepsilon$.

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

Case: ε-VAR. Then $\hat{e} = y$ and either $y = x$ or $y \neq x$. If $y \neq x$. Then $[\hat{v}/x]y = y$ and $\hat{\Gamma} \vdash y : \hat{\tau} \texttt{ with } \varnothing$. Therefore $\hat{\Gamma} \vdash [\hat{v}/x]y : \hat{\tau} \texttt{ with } \varnothing$. Otherwise $y = x$. By inversion on ε-VAR, the typing judgement from the theorem assumption is $\hat{\Gamma}, x : \hat{\tau}' \vdash x : \hat{\tau}' \texttt{ with } \varnothing$. Since $[\hat{v}/x]y = \hat{v}$, and by assumption $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}' \texttt{ with } \varnothing$, then $\hat{\Gamma} \vdash [\hat{v}/x]x : \hat{\tau}' \texttt{ with } \varnothing$.

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

Case: ε-APP By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e}_1 : \hat{\tau}_2 \rightarrow_{\varepsilon_3} \hat{\tau}_3 \texttt{ with } \varepsilon_A$ and $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e}_2 : \hat{\tau}_2 \texttt{ with } \varepsilon_B$, where $\varepsilon = \varepsilon_A \cup \varepsilon_B \cup \varepsilon_3$ and $\hat{\tau} = \hat{\tau}_3$. By inductive assumption, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_1 : \hat{\tau}_2 \rightarrow_{\varepsilon_3} \hat{\tau}_3 \texttt{ with } \varepsilon_A$ and $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_2 : \hat{\tau}_2 \texttt{ with } \varepsilon_B$. By ε-APP we have $\hat{\Gamma} \vdash ([\hat{v}/x]\hat{e}_1)([\hat{v}/x]\hat{e}_2) : \hat{\tau}_3 \texttt{ with } \varepsilon_A \cup \varepsilon_B \cup \varepsilon_3$. By simplifying and applying the definition of `substitution`, this is the same as $\hat{\Gamma} \vdash [\hat{v}/x](\hat{e}_1\hat{e}_2) : \hat{\tau} \texttt{ with } \varepsilon$.

Case: ε-OPERCALL By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e}_1 : \{\bar{r}\} \texttt{ with } \varepsilon_1$, where $\varepsilon = \varepsilon_1 \cup \{r.\pi \mid r.\pi \in \bar{r} \times \Pi\}$ and $\hat{\tau} = \{\bar{r}\}$. By applying the inductive assumption, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_1 : \{\bar{r}\} \texttt{ with } \varepsilon_1$. Then by ε-OPERCALL, $\hat{\Gamma} \vdash ([\hat{v}/x]\hat{e}_1).\pi : \{\bar{r}\} \texttt{ with } \varepsilon_1 \cup \{r.\pi \mid r.\pi \in \bar{r} \times \Pi\}$. By simplifying and applying the definition of `substitution`, this is the same as $\hat{\Gamma} \vdash [\hat{v}/x](\hat{e}_1.\pi) : \hat{\tau} \texttt{ with } \varepsilon$.

Case: ε-SUBSUME By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}_2 \texttt{ with } \varepsilon_2$, where $\hat{\tau}_2 <: \hat{\tau}$ and $\varepsilon_2 \subseteq \varepsilon$. By inductive hypothesis, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}_2 \texttt{ with } \varepsilon_2$. Then by ε-SUBSUME we get $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau} \texttt{ with } \varepsilon$.

Case: ε-MODULE Then $\hat{\Gamma}, x : \hat{\tau}' \vdash \texttt{import}(:) = \mathit{annot} \texttt{ in } (\tau, \varepsilon) \texttt{ with } \varepsilon \cup \varepsilon_1$. By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}_1 \texttt{ with } \varepsilon_1$. By inductive assumption, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}_1 \texttt{ with } \varepsilon_1$. Then by ε-MODULE we have $\hat{\Gamma} \vdash \texttt{import}(:) = \mathit{annot} \texttt{ in } (\tau, \varepsilon) \texttt{ with } \varepsilon \cup \varepsilon_1$. □

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

Lemma 15. *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.

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

Case: $\hat{\tau} = \hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2$, $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ It is sufficient to show $\hat{\tau}_2 <: \texttt{annot}(\texttt{erase}(\hat{\tau}_2), \varepsilon)$ and $\texttt{annot}(\texttt{erase}(\hat{\tau}_1), \varepsilon) <: \hat{\tau}_1$, because the result will hold by S-EFFECTS. To achieve this we shall inductively apply **lemma 2** to $\hat{\tau}_2$ and **lemma 3** to $\hat{\tau}_1$.

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

From $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon$ we have $\texttt{ho-effects}(\hat{\tau}_1) \cup \varepsilon' \cup \texttt{effects}(\hat{\tau}_2) \subseteq \varepsilon$ and therefore $\texttt{ho-effects}(\hat{\tau}_1) \subseteq \varepsilon$. From $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ we have $\texttt{ho-safe}(\hat{\tau}_1, \varepsilon)$. Therefore we can apply **lemma 3** to $\hat{\tau}_1$.

Case: $\hat{\tau} = \hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2$, $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$, $\texttt{safe}(\hat{\tau}, \varepsilon)$ It is sufficient to show $\texttt{annot}(\texttt{erase}(\hat{\tau}_2), \varepsilon) <: \hat{\tau}_2$ and $\hat{\tau}_1 <: \texttt{annot}(\texttt{erase}(\hat{\tau}_1), \varepsilon)$, because the result will hold by S-EFFECTS. To achieve this we shall inductively apply **lemma 3** to $\hat{\tau}_2$ and **lemma 2** to $\hat{\tau}_1$.

From $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ we have $\texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2) \subseteq \varepsilon$ and therefore $\texttt{ho-effects}(\hat{\tau}_2) \subseteq \varepsilon$. From $\texttt{safe}(\hat{\tau}, \varepsilon)$ we have $\texttt{safe}(\hat{\tau}_2, \varepsilon)$. Therefore we can apply **lemma 3** to $\hat{\tau}_2$.

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

□

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

- $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}\ \texttt{with}\ \varnothing$
- $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e : \tau$
- $\varepsilon = \texttt{effects}(\hat{\tau})$
- $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$

Then $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$.

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 ε-VAR we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash x : \hat{\tau}\ \texttt{with}\ \varnothing$. First note that $\texttt{annot}(x, \varepsilon) = x$ in this case. Therefore $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash \texttt{annot}(\texttt{erase}(x), \varepsilon) : \hat{\tau}\ \texttt{with}\ \varnothing$. We know by assumption that $\texttt{effects}(\hat{\tau}) = \varepsilon$ and $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$. Applying **Lemma 2** we know $\hat{\tau} <: \texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon)$. Lastly, by ε-SUBSUME we have $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash \texttt{annot}(\texttt{erase}(x), \varepsilon) : \texttt{annot}(\texttt{erase}(x), \varepsilon)\ \texttt{with}\ \varepsilon \cup \texttt{effects}\ (\texttt{annot}(\Gamma, \varepsilon))$.

Subcase 2: $x \neq y$. Then $x : \tau \in \Gamma$. Together with the definition $\texttt{annot}(x, \varepsilon) = x$, we know $x : \texttt{annot}(\tau, \varepsilon) \in \texttt{annot}(\Gamma, \varepsilon)$. By ε-VAR we have $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(x, \varepsilon) : \texttt{annot}(\tau, \varepsilon)\ \texttt{with}\ \varnothing$. Lastly, by ε-SUBSUME we have $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash \texttt{annot}(\texttt{erase}(x), \varepsilon) : \texttt{annot}(\texttt{erase}(x), \varepsilon)\ \texttt{with}\ \varepsilon \cup \texttt{effects}\ (\texttt{annot}(\Gamma, \varepsilon))$.

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 $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash r : \{r\}\ \texttt{with}\ \varnothing$. By ε-SUBSUME, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash r : \{r\}\ \texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon))$.

Case: T-ABS Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash \lambda x : \tau_1.e_{body} : \tau_1 \rightarrow \tau_2$. By inversion, we get the sub-derivation $\Gamma, y : \texttt{erase}(\hat{\tau}), x : \tau_1 \vdash e_2 : \tau_2$. By definition, $\texttt{annot}(e, \varepsilon) = \texttt{annot}(\lambda x : \tau_1.e_2, \varepsilon) = \lambda x : \texttt{annot}(\tau_1, \varepsilon).\texttt{annot}(e_2, \varepsilon)$ and $\texttt{annot}(\tau, \varepsilon) = \texttt{annot}(\tau_1 \rightarrow \tau_2, \varepsilon) = \texttt{annot}(\tau_1, \varepsilon) \rightarrow_{\varepsilon} \texttt{annot}(\tau_2, \varepsilon)$.

To apply the inductive assumption to e_2 we use the unlabelled context $\Gamma, x : \tau_1$. The inductive assumption tells us $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau}, x : \texttt{annot}(\tau_1, \varepsilon) \vdash \texttt{annot}(e_2, \varepsilon) : \texttt{annot}(\tau_2, \varepsilon)\ \texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma, \varepsilon)) \cup \texttt{effects}(\texttt{annot}(\tau_1, \varepsilon))$. Call this last effect-set ε'. By ε-ABS, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \lambda x : \texttt{annot}(\tau_1, \varepsilon).\texttt{annot}(e_2, \varepsilon) : \texttt{annot}(\hat{\tau}_1) \rightarrow_{\varepsilon'} \texttt{annot}(\hat{\tau}_2)\ \texttt{with}\ \varnothing$. Then by ε-SUBSUME, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\hat{\tau}_1) \rightarrow_{\varepsilon} \texttt{annot}(\hat{\tau}_2)\ \texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma), \varepsilon)$.

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

Case: T-OPERCALL Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1.\pi : \texttt{Unit}$. By inversion we get the sub-derivation $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1 : \{\bar{r}\}$. 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 20 (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-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 $e_A = \hat{e}_1\ \hat{e}_2$ and $\hat{e}_1 : \hat{\tau}_2 \to_\varepsilon \hat{\tau}_3 \texttt{ with } \varepsilon_1$ and $\hat{\Gamma} \vdash \hat{e}_2 : \hat{\tau}_2 \texttt{ with } \varepsilon_2$.

Subcase: E-APP1. Todo.

Subcase: E-APP2. Todo.

Subcase: E-APP3. Then $(\lambda x : \hat{\tau}_2.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x]\hat{e} \mid \varnothing$. By inversion on the typing rule for $\lambda x : \hat{\tau}_2.\hat{e}$ we know $\Gamma, x : \hat{\tau}_2 \vdash \hat{e} : \hat{\tau}_3 \texttt{ with } \varepsilon_3$. By canonical forms, $\varepsilon_2 = \varnothing$ because $\hat{e}_2 = \hat{v}_2$ is a value. Then by the substitution lemma, $\hat{\Gamma} \vdash [\hat{v}_2/x]\hat{e} : \hat{\tau}_3 \texttt{ with } \varepsilon_3$. By canonical forms, $\varepsilon_1 = \varepsilon_2 = \varnothing = \varepsilon_C$. Therefore $\varepsilon_A = \varepsilon_3 = \varepsilon_B \cup \varepsilon_C$.

Case: ε-OPERCALL.

Subcase: E-OPERCALL1.

Subcase: Otherwise the reduction rule used was E-OPERCALL2 and $v_1.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}$. By canonical forms, $\hat{\Gamma} \vdash v_1 : \texttt{unit with } \{r.\pi\}$. Also, $\hat{\Gamma} \vdash \texttt{unit} : \texttt{Unit with } \varnothing$. Then $\tau_B = \tau_A$. Also, $\varepsilon_C \cup \varepsilon_B = \{r.\pi\} = \varepsilon_A$.

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 21 (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 $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.*

Proof. If $\hat{e}_A$ is not a value then the reduction exists by the progress theorem. The rest follows by the preservation theorem. □

Theorem 22 (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 $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.*

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 $\hat{e}_A \longrightarrow^* \hat{e}_C \mid \varepsilon'$ 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, $\hat{\Gamma} \vdash \hat{e}_C : \hat{\tau}_C\ \texttt{with}\ \varepsilon_C$ and $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B\ \texttt{with}\ \varepsilon_B$. By inductive assumption, $\hat{\tau}_C <: \hat{\tau}_A$ and $\hat{\varepsilon}_C \cup \varepsilon' \subseteq \varepsilon_A$. By single-step soundness, $\hat{\tau}_B <: \hat{\tau}_C$ and $\hat{\varepsilon}_B \cup \varepsilon'' \subseteq \varepsilon_C$. Then by transitivity, $\hat{\tau}_B <: \hat{\tau}$ and $\hat{\varepsilon}_B \cup \varepsilon' \cup \varepsilon'' = \varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$. □

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

[4] Dennis, J. B., and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM 9*, 3 (1966), 143–155.

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

[6] 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.

[7] Liskov, B. Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)* (New York, NY, USA, 1987), OOPSLA '87, ACM, pp. 17–34.

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

[9] Maffeis, S., Mitchell, J. C., and Taly, A. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy* (2010).

[10] Miller, M., Yee, K.-P., and Shapiro, J. Capability myths demolished. Tech. rep., 2003.

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

[12] Nielson, F., and Nelson, H. R. Type and Effect Systems. pp. 114–136.

[13] 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.

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

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