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. One of these is the *principle of least authority*: that software components should only have access to the information and resources necessary for their purpose [11]. 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 **[?]**.

Matters get complicated when a program is contains untrustworthy components. Such cases may arise in a development environment which adheres to *code ownership*, whereby groups of developers may function as particular experts over certain components **[?]**. When they interact with code sourced from outside their domain of expertise, they may make false assumptions or violate the internal constraints of other components. Applications may 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.

When a codebase has untrustworthy code it may be impossible or infeasible for developers to verify that it adequately enforces separation of privileges and POLA. Often they may not have access to the original source code. This leaves developers to make a judgement call on whether this untrusted code should be used or executed based on the type interface and accompanying verbal contracts.

One approach to privilege separation is the *capability* model. A capability is an unforgeable token granting its bearer permission to perform some operation [3]. 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. Miller has identified the ways in which capabilities should proliferate to encourage *robust composition* — a set of ideas summarised as "only connectivity begets connectivity" [7]. In his paradigm, the reference graph of a program is the same as the access graph. This eliminates *ambient authority* — a pervasive enemy in determining by interface what privileges a component might exercise. Building on these ideas, Maffeis et. al. formalised *capability-safety* of a language, showing a subset of Caja (a JavaScript implementation) meets this notion [5].

While capabilities adequately separate privileges, understanding the way in which those privileges are exercised has received less attention. This area falls under the domain of effect systems, which extends type systems to include intensional information about the way in which a program executes [8]. For example, a logger's log method may have `append` as its effect, but a sloppy or malicious implementation may incur extra effects, such as `write` or `close`. This suggests the logger may be doing more than just logging, and knowing this guides the developer to a more informed decision about whether to use this particular implementation.

One obstacle to the adoption of effect systems is their verbosity. An effect system such as the Talpin-Jouvelot system requires the annotation of all values in a program **[cite]**. This requires the developer to be aware, at all points, of what effects are happening and where. Minor alterations to the signatures and effects of a component 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 programming, affecting the usability of effect 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. This paper suggests that capability-safety permits an effect system with minimal overhead.

This paper's contribution is to develop an extension to the simply-typed lambda calculus $\lambda^{\rightarrow}$ called the epsilon calculus $\lambda^{\rightarrow}_{\varepsilon}$. $\lambda^{\rightarrow}_{\varepsilon}$ introduces a new construct which selects those capabilities used in a peice of unlabelled code. A sound inference can be made about the unlabelled code by inspecting the type signatures of those functions in scope at the point of introduction. This is made possible by the restrictions imposed on the use and exchange of capabilities.

Chapter 2 covers some background information on capabilities and programming language semantics. It also establishes the various conventions used throughout. We identify some of the benefits obtained by capabilities and effects, and some of the drawbacks we set out to solve.

Chapter 3 introduces static and dynamic rules for $\lambda^{\rightarrow}_{\varepsilon}$, developing and proving a formulation of soundness appropriate for the type-and-effect discipline.

Chapter 4 shows how $\lambda^{\rightarrow}_{\varepsilon}$ might solve these drawbacks, and try to convince the reader that $\lambda^{\rightarrow}_{\varepsilon}$ can be implemented in existing capability-safe languages in a routine manner.

Chapter 2

Background

In this section we cover some of the necessary concepts and existing work informing this paper. No prior knowledge is assumed.

2.1 Formal Semantics

We will consider a programming language as three sets of rules: the grammar, the static rules, and the dynamic rules. To illustrate each we give rules for a toy language that evaluates basic arithmetic operations on $\mathbb{N}$.

The grammar specifies what strings are syntactically legal in the language. A grammar 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.

$$
\begin{array}{lllr}
e & ::= & & \mathit{exprs}: \\
 & | & x & \mathit{variable} \\
 & | & e + e & \mathit{addition} \\
 & | & l & \mathit{integer\ constant}
\end{array}
\qquad
\begin{array}{lllr}
\tau & ::= & & \mathit{types}: \\
 & | & \texttt{Int} & \\
 & & & \\
\Gamma & ::= & & \mathit{contexts}: \\
 & | & \varnothing & \\
 & | & \Gamma, x : \tau &
\end{array}
$$

Figure 2.1: Grammar for arithmetic expressions.

The static rules specify the type system and other constraints on terms with certain *well-behavedness* properties. In our case, we're interested in what makes a program *well-typed*, which is to say that execution of the program never gets *stuck* due to type-errors. For example, a well-typed program will never try to add two booleans, because addition only makes sense on numbesr. A well-typed program will never try to evaluate an undefined variable, because variables must be defined.

Static rules are specified by *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. An application of an inference rule is called a *judgement*. Judgements take place in typing contexts, denoted by Γ, which map variables to types. A basic judgement like $\Gamma \vdash e : \tau$ means that executing e will result in a term of type τ (if it terminates). The contents of Γ are specified as a comma-separated sequence of variable-type pairs. The order is irrelevant. A consequence of this convention is that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash e : \tau$, where Γ is contained in Γ'. When a judgement can be proven from the empty context we leave the left of the turnstile blank, as in $\vdash e : \tau$.

Though our arithmetic language has no subtyping, most interesting languages do. This judgement is written $\tau_1 <: \tau_2$ and it means that values of τ_1 may be provided anywhere instances of τ_2 are expected. Effect systems have another judgement, which ascribes a type and set of effects to an expression. We shall cover this in greater detail later.

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

$$\frac{}{\Gamma, x : \texttt{Int} \vdash x : \texttt{Int}}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}}\ (\text{T-Add})$$

Figure 2.2: Inference rules for typing arithmetic expressions.

The dynamic semantics specifies what is the meaning of a legal term. There are different flavorus of dynamic semantics, but the one we use is called *small-step semantics*. This is a set of inference rules specifying how a program is executed. A single application of one of these rules is called a *reduction*.

$\boxed{e \longrightarrow e}$

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ (\text{E-Add1}) \quad \frac{e_2 \longrightarrow e_2'}{l_1 + e_2 \longrightarrow l_1 + e_2'}\ (\text{E-Add2}) \quad \frac{l_1 + l_2 = l_3}{l_1 + l_2 \longrightarrow l_3}\ (\text{E-Add3})$$

Figure 2.3: Inference rules for reducing arithmetic expressions.

Almost all type systems in which we are interested are *sound*. Soundness is a property that holds between the static and dynamic rules of a language, which says that if a program e is considered well-typed by the static rules, then its reduction under the dynamic rules will never produce a runtime type-error. The exact definition of soundness depends on the semantics under consideration, but it is often split into two parts: progress and preservation. The progress theorem states every term, except those of a particular category called values, can always be reduced by applying some dynamic rule.

The preservation theorem is that programs remain well-typed under reduction. Adequate formulations of these two theorems for the language under consideration give us soundness.

In the language of arithmetic expressions, progress and preservation would be the following:

Theorem 1 (Progress). *If* $\Gamma \vdash e : \texttt{Int}$ *and* e *is not an integer constant, then* $e \to e'$.

Theorem 2 (Preservation). *If* $\Gamma \vdash e : \texttt{Int}$ *and* $e \to e'$ *then* $\Gamma \vdash e' : \texttt{Int}$.

These theorems can be proven by structural induction on the typing judgement $\Gamma \vdash e : \texttt{Int}$. This is a common proof technique in formal semantics. A more thorough explanation of this sort of induction can be found in TAPL [10, p. 31].

2.2 Capability Safety

A *capability* is a unique, unforgeable reference, giving its bearer permission to perform some operation [3]. 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 [7]:

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, $\lambda \texttt{x} : \texttt{Int}.\ \texttt{x}$ is pure, while $\lambda \texttt{f} : \texttt{File}.\ \texttt{f.log}(\text{"}\texttt{error message}\text{"})$ 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 [6]. 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.

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

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

```java
import java.util.List;

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

Figure 2.4: `Main` exercises ambient authority over a `File`.

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

2.3 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 [9], 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 [4].

2.4 Effect Systems

Some languages extend the notion of a type system to a *type-and-effect system*. Effects describe intensional information about the way in which a program executes [8]. A judgement like $\Gamma \vdash \texttt{e} : \tau \ ! \ \{\texttt{File.write}\}$ can be interpreted as meaning that execution of e will result in a value of type τ (if it halts), and during execution might perform the `write` effect on a `File`. In theteffects literature, `File` would be called the region and `write` the kind of effect. We instead called them *resource* and *operation*, befitting the capability focus.

Chapter 3

Semantics

3.1 Grammar

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

$$
\begin{array}{lll}
\varepsilon & ::= & \hfill \textit{effects}: \\
 & \mid \quad \{\overline{r.\pi}\} & \\
\\
\tau & ::= & \hfill \textit{types}: \\
 & \mid \quad \{\bar{r}\} & \\
 & \mid \quad \tau \to \tau & \\
\\
\hat{\tau} & ::= & \hfill \textit{labelled types}: \\
 & \mid \quad \{\bar{r}\} & \\
 & \mid \quad \hat{\tau} \to_{\varepsilon} \hat{\tau} & \\
\\
\Gamma & ::= & \hfill \textit{type ctx}: \\
 & \mid \quad \varnothing & \\
 & \mid \quad \Gamma, x : \tau & \\
\\
\hat{\Gamma} & ::= & \hfill \textit{labelled type ctx}: \\
 & \mid \quad \varnothing & \\
 & \mid \quad \hat{\Gamma}, x : \hat{\tau} &
\end{array}
$$

Figure 3.1: Effect calculus.

The effect calculus is based on the simply-typed lambda calculus $\lambda^{\to}$. 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.

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 $\texttt{File.write}$ 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 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 $\texttt{File.write}(\text{“}\texttt{mymsg}\text{”})$. Because $\lambda_{\varepsilon}^{\rightarrow}$ 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 $\hat{e}$.

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

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

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

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

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

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

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

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

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

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

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

Figure 3.2: Annotation functions.

of code $\texttt{annot}(e, \varepsilon) = \hat{e}$ by annotating every function with ε. In the reverse direction, given some labelled code $\hat{e}$ we can produce an unlabelled piece of code $\texttt{erase}(\hat{e}) = e$ 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, $\texttt{annot}(\lambda l : \texttt{Int} \rightarrow_{File.read} \texttt{Int}.\ l\ 1, \varnothing)$ 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, $\texttt{effects}(\hat{e})$ and $\texttt{ho-effects}(\hat{e})$, 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.

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

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

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

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

Figure 3.3: Effect functions.

$\texttt{effect}(\hat{e})$ is the set of effects for which $\hat{e}$ has direct authority, while `ho-effects` is the 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.

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

$[\hat{v}/y]x = \hat{v}$, 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 3.4: 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 [10, p. 71]. This elides some tedious bookkeeping. Consequently, we shall assume variables are (re-)named in this way to avoid accidental capture.

3.2 Static Rules

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

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

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

Figure 3.5: 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 $\lambda^{\to}$. 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.

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

$$\frac{}{\texttt{safe}(\{\bar{r}\}, \varepsilon)} \text{ (SAFE-RESOURCE)} \qquad \frac{}{\texttt{safe}(\texttt{Unit}, \varepsilon)} \text{ (SAFE-UNIT)}$$

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

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

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

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

Figure 3.6: 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.**

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

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

$$\frac{\hat{\Gamma}, x : \hat{\tau}_2 \vdash \hat{e} : \hat{\tau}_3 \texttt{ with } \varepsilon_3}{\hat{\Gamma} \vdash \lambda x : \tau_2 . \hat{e} : \hat{\tau}_2 \rightarrow_{\varepsilon_3} \hat{\tau}_3 \texttt{ with } \varnothing}\ (\varepsilon\text{-}\textsc{Abs}) \quad \frac{\hat{\Gamma} \vdash \hat{e}_1 : \hat{\tau}_2 \rightarrow_{\varepsilon} \hat{\tau}_3 \texttt{ with } \varepsilon_1 \quad \hat{\Gamma} \vdash \hat{e}_2 : \hat{\tau}_2 \texttt{ with } \varepsilon_2}{\hat{\Gamma} \vdash \hat{e}_1 \hat{e}_2 : \hat{\tau}_3 \texttt{ with } \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon}\ (\varepsilon\text{-}\textsc{App})$$

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

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

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

Figure 3.7: Type-with-effect judgements.

$\lambda_{\varepsilon}^{\rightarrow}$ 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 $\{\texttt{File.write}, \texttt{Socket.write}\}$.

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 3.8: We cannot statically determine which branch will execute, so the safe approximation for $\texttt{getResource(boolVal).write}$ 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.

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

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

Figure 3.9: 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.

3.3 Dynamic Rules

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

$$\frac{\hat{e}_1 \longrightarrow \hat{e}_1' \mid \varepsilon}{\hat{e}_1\hat{e}_2 \longrightarrow \hat{e}_1'\hat{e}_2 \mid \varepsilon} \text{ (E-APP1)} \qquad \frac{\hat{e}_2 \longrightarrow \hat{e}_2' \mid \varepsilon}{\hat{v}_1\hat{e}_2 \longrightarrow \hat{v}_1\hat{e}_2' \mid \varepsilon} \text{ (E-APP2)} \qquad \frac{}{(\lambda x : \hat{\tau}.\hat{e})\hat{v}_2 \longrightarrow [\hat{v}_2/x]\hat{e} \mid \varnothing} \text{ (E-APP3)}$$

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

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

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

Figure 3.10: 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 $\lambda_{\varepsilon}^{\rightarrow}$ 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.

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

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

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

Figure 3.11: Multi-step reductions.

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.

3.4 Soundness

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

Theorem 3 (Soundness). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ $\texttt{with}$ ε_A and $\hat{e}_A$ is not a value, then $e_A \longrightarrow e_B \mid \varepsilon$, where $\hat{\Gamma} \vdash e_B : \hat{\tau}_B$ $\texttt{with}$ 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 $\lambda^{\to}$ 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 1 (Canonical Forms). *The following are true:*

- *If $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}$ $\texttt{with}$ 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 4 (Progress). *If $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ $\texttt{with}$ ε and $\hat{e}$ is not a value, then $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon$.*

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

[1]We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation[10, p. 39].

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

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 3 (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 4. *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 5. *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 5 (Preservation). *If* $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ *and* $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$, *then* $\hat{\tau}_B <: \hat{\tau}_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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

Case: ε-APP Then $e_A = \hat{e}_1\hat{e}_2$ and $\hat{e}_1 : \hat{\tau}_2 \rightarrow_\varepsilon \hat{\tau}_3 \texttt{ with } \varepsilon_1$ and $\hat{\Gamma} \vdash \hat{e}_2 : \hat{\tau}_2 \texttt{ with } \varepsilon_2$. If the reduction rule used was E-APP1 or E-APP2, then the result follows by applying the inductive hypothesis to $\hat{e}_1$ 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 $\Gamma, x : \hat{\tau}_2 \vdash \hat{e} : \hat{\tau}_3 \texttt{ with } \varepsilon_3$. 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. Then $e_A = e_1.\pi$ and $\hat{\Gamma} \vdash e_1 : \{\bar{r}\} \texttt{ with } \varepsilon_1$. If the reduction rule used was E-OPERCALL1 then the result follows by applying the inductive hypothesis to $\hat{e}_1$.

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

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

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

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

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

Theorem 6 (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. □

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 7 (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. 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 $\texttt{unit} \stackrel{\text{def}}{=} \lambda x : \varnothing . x$ and $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \rightarrow_{\varnothing} \varnothing$. Note that because there is no empty resource literal, `unit` cannot be applied to anything. Furthermore, $\vdash \texttt{unit} : \texttt{Unit with } \varnothing$, by ε-ABS, so any context can make this type judgement.

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

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

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

Figure 4.1: Derived `Unit` rules.

4.1.2 Let

The expression $\texttt{let } x = \hat{e}_1 \texttt{ in } \hat{e}_2$ first binds the value $\hat{e}_1$ to the name x and then evaluates $\hat{e}_2$. We can generalise by allowing $\hat{e}_1$ to be a non-value, in which case it must first be reduced to a value. If $\Gamma \vdash \hat{e}_1 : \hat{\tau}_1$, then $\texttt{let } x = \hat{e}_1 \texttt{ in } \hat{e}_2 \stackrel{\text{def}}{=} (\lambda x : \hat{\tau}_1 . \hat{e}_2)\hat{e}_1$. Note that if $\hat{e}_1$ is a non-value, we can reduce the `let` by E-APP2. If $\hat{e}_1$ is a value, we may apply E-APP3, which binds $\hat{e}_1$ to x in $\hat{e}_2$. This is fundamentally a lambda application, so it can be typed using ε-APP (or T-APP, if the terms involved are unlabelled).

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

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

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

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

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

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

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

Figure 4.2: Derived `let` rules.

4.1.3 Conditionals

4.1.4 Tuples

We need tuples to import multiple names.

Appendix A

Proofs

Lemma 6 (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 8 (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.

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 $\hat{\Gamma} \vdash v_1 : \{r\}$ `with` $\varnothing$. Then $r.\pi \longrightarrow$ `unit` $\mid \{r.\pi\}$ by E-OPERCALL2.

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

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

Lemma 7 (Substitution). *If* $\hat{\Gamma}, x : \hat{\tau}' \vdash e : \hat{\tau}$ `with` ε *and* $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}'$ `with` $\varnothing$ *then* $\hat{\Gamma} \vdash [\hat{v}/x]e : \hat{\tau}$ `with` ε.

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

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

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

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

Case: ε-OPERCALL By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e}_1 : \{\bar{r}\}$ `with` ε_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, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e}_1 : \{\bar{r}\}$ `with` ε_1. Then by ε-OPERCALL, $\hat{\Gamma} \vdash ([\hat{v}/x]\hat{e}_1).\pi : \{\bar{r}\}$ `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 $\hat{\Gamma} \vdash [\hat{v}/x](\hat{e}_1.\pi) : \hat{\tau}$ `with` ε.

Case: ε-SUBSUME By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}_2$ `with` ε_2, where $\hat{\tau}_2 <: \hat{\tau}$ and $\varepsilon_2 \subseteq \varepsilon$. By inductive hypothesis, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}_2$ `with` ε_2. Then by ε-SUBSUME we get $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}$ `with` ε.

Case: ε-MODULE Then $\hat{\Gamma}, x : \hat{\tau}' \vdash$ `import`$(:) =$*annot* `in` (τ, ε) `with` $\varepsilon \cup \varepsilon_1$. By inversion we know $\hat{\Gamma}, x : \hat{\tau}' \vdash \hat{e} : \hat{\tau}_1$ `with` ε_1. By inductive assumption, $\hat{\Gamma} \vdash [\hat{v}/x]\hat{e} : \hat{\tau}_1$ `with` ε_1. Then by ε-MODULE we have $\hat{\Gamma} \vdash$ `import`$(:) =$*annot* `in` (τ, ε) `with` $\varepsilon \cup \varepsilon_1$. □

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

Lemma 9. *If* `ho-effects`$(\hat{\tau}) \subseteq \varepsilon$ *and* `safe`$(\hat{\tau}, \varepsilon)$ *then* `annot(erase`$(\hat{\tau}), \varepsilon) <: \hat{\tau}$.

Proof. By simultaneous induction.

Case: $\hat{\tau} = \{\bar{r}\}$ Then $\hat{\tau} =$ `annot(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 10 (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 $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash \lambda x : \tau_1.e_{body} : \tau_1 \to \tau_2$. 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 $\texttt{annot}(\tau, \varepsilon) = \texttt{annot}(\tau_1 \to \tau_2, \varepsilon) = \texttt{annot}(\tau_1, \varepsilon) \to_\varepsilon \texttt{annot}(\tau_2, \varepsilon)$.

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 $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \lambda x : \texttt{annot}(\tau_1, \varepsilon).\texttt{annot}(e_2, \varepsilon) : \texttt{annot}(\hat{\tau}_1) \to_{\varepsilon'} \texttt{annot}(\hat{\tau}_2)\ \texttt{with}\ \varnothing$. Then by ε-SUBSUME, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\hat{\tau}_1) \to_\varepsilon \texttt{annot}(\hat{\tau}_2)\ \texttt{with}\ \varepsilon \cup \texttt{effects}(\texttt{annot}(\Gamma), \varepsilon)$.

Case: T-APP Then $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1\ e_2 : \tau_3$, where $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_1 : \tau_2 \to \tau_3$ and $\Gamma, y : \texttt{erase}(\hat{\tau}) \vdash e_2 : \tau_2$. By applying the inductive assumption to e_1 and e_2, we get $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e_1, \varepsilon) : \texttt{annot}(\tau_1, \varepsilon)\ \texttt{with}\ \varepsilon$ 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, $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e_1, \varepsilon) : \texttt{annot}(\tau_2, \varepsilon) \to_\varepsilon \texttt{annot}(\tau_3, \varepsilon)\ \texttt{with}\ \varepsilon$. 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 9 (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 10 (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 11 (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] DENNIS, J. B., AND VAN HORN, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM 9*, 3 (1966), 143–155.

[4] 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.

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

[6] MILLER, M., YEE, K.-P., AND SHAPIRO, J. Capability myths demolished. Tech. rep., 2003.

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

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

[9] 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.

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

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