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

An abstract of fewer than 500 words must be included.

Acknowledgments

Contents

Chapter 1

Introduction

Hello this is the intro.

Chapter 2

Background

In this section we briefly cover some of the necsesary prerequisites and existing work. No prior knowledge is assumed.

2.1 Formal Semantics

We will consider a programming language as three sets of rules.

The grammar specifies what strings are legal terms within 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.

$$
\begin{array}{rcll}
 & & & \tau ::= \quad \textit{types}: \\
e & ::= & \textit{exprs}: & \mid \quad \texttt{Int} \\
 & \mid \quad x & \textit{variable} & \\
 & \mid \quad e + e & \textit{addition} & \Gamma ::= \quad \textit{contexts}: \\
 & \mid \quad l & \textit{integer constant} & \mid \quad \varnothing \\
 & & & \mid \quad \Gamma, x : \tau
\end{array}
$$

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. Static rules are specified with a set of *inference rules*. An inference rule is given as a set of premises above a diving 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, which map variables to types. A basic judgement, like "e has type τ", would be written $\Gamma \vdash e : \tau$. When the context is empty it is customary to write $\vdash e : \tau$.

Most languages have some form of subtyping. 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.

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

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

Figure 2.2: Inference rules for typing arithmetic expressions.

The dynamic semantics specifies what the meaning of a legal term is. There are different approaches, but the one we take is to give a 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 of a language is a property between its static and dynamic rules, which essentially says that if a program e is considered well-typed by the static rules, then its reduction under the dynamic rules will never get stuck. Soundness is often split into two parts: progress and preservation. The progress theorem is that 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 gives us soundness.

2.2 Module Systems

The division of a codebase into logical modules is a common technique in many languages to help developers write code that is re-usable, easy to test and debug, and safer.

2.3 Capability Safety

A capability is a unique, unforgeable reference, giving its bearer permission to perform some operation [6]. A piece of code S has *authority* over a capability if it can directly

invoke the operations endowed by a capability 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 access to C).

Authority may only proliferate in the following ways [3]:

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".

Atomic capabilities are *resources*. A capability is any object or function with authority over a resource, or over another 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 of S over C where S has not explicitly declared its authority [2]. Figure 2.4. gives an example in Java, where a malicious implementation of `List.add` attempts to overwrite the user's `.bashrc` file. From the persective of `Main`, inspecting the signatures of all imports is not sufficient to determine what authority is being exercised. Determining this would require one to look at source code outside of `Main`, which contravenes code ownership.

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

2.4 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 [4]. A judgement like $\Gamma \vdash$ `e` $: \tau \,!\, \{$`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 the effects literature, `File` would be called the region and `write` the kind of effect. We instead called them *resource* and *operation*, befitting the capability focus.

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

Chapter 3

Semantics

3.1 Grammar

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

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

Figure 3.1: 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 $\{\bar{r}\}$. Resources are drawn from a fixed set R of variables. They describe those initial capabilites from which all others are derived. 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 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 $(\texttt{File}, \texttt{write})$. A set of effects is denoted by ε. Effects and operations look the same, but should be distinguished. An effect is some intensional property describing the way in which a computation occurs; an operation is the runtime invocation of an effect.

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 our theory is only concerned with the use and propagation of effects, and not their particular semantics, 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 $\hat{e}$.

A labelled term $\hat{e}$ is *deeply* labelled. This means every subterm is also labelled. An unlabelled term e is deeply unlabelled. The only exception to this rule is the `import` expression, which is the only way to compose labelled and unlabelled code. `import` nests unlabelled code inside labelled code.

It is not possible to nest labelled code inside unlabelled code. Intuitively, this restriction is made because of **reasons...**

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.

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, 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`

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

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

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

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

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

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

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

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

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

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

Figure 3.2: Annotation functions.

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.

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. $\texttt{effect}(\hat{e})$ is the set of effects for which $\hat{e}$ is responsible: those which it has direct authority to use, or those which it returns from a function. `ho-effects` returns the set of effects that $\hat{e}$ may use, but which have been 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 $\texttt{f} : \{\texttt{File}\} \to_{\texttt{File.read}} \texttt{Int}$. 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`

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

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

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

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

Figure 3.3: Effect functions.

literal from some outside source. Therefore, every possible effect on `File` is a higher-order effect.

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

$[\hat{v}/y]x = \hat{v}$, 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 3.4: 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 introduce arbitrary effects into a piece of code as the result of substitition.

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 [5, 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

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 essentially the

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

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

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

Figure 3.5: Typing judgements in the epsilon calculus.

same as T-VAR. T-OPERCALL is the rule for typing expressions of the form $e_1.\pi$. Such an expression is well-typed if e_1 types to some valid resource, and π is a valid 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 $\hat{\tau}$ was not expecting. 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.**

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

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

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

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

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

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

Figure 3.7: Type-with-effect judgements.

The epsilon calculus has a new kind of judgement: $\Gamma \vdash \hat{e} : \hat{\tau}$ `with` ε can be read as saying that $\hat{e}$, if it halts, will produce an expression of type $\hat{\tau}$ and incur at most the set of effects ε. This judgement gives a conservative approximation as to what will happen; it is not necessarily tight.

The simple rules are those which operate on values. They type as having no effect. This is because, although a function and a resource literal both capture capabilities, you must do something with them (apply the function, or operate on the resource) in order to produce an effect at runtime.

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 evaluated. This is the set with which the lambda's arrow-type is annotated.

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 $\{$`File.write`, `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 `getResource(boolVal).write` is $\{\texttt{File.write}, \texttt{Socket.write}\}$.

The most interesting rule is ε-Module. 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 elimiantes the problem of ambient authority, because the only authority exercised in e is that which is explicitly selected by the interface $\hat{\tau}$ of the module. If we allowed it to type with any extra information in context, we might not know that the unlabelled code isn't widening the effects captured by the labelled code it imports.

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 the capabilities to which they have access.

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 it is a sound approximation.

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

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

Figure 3.9: Subtyping judgements in the epsilon calculus.

In addition to the usual subtyping rules from $\lambda^{\to}$ 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 $\lambda^{\to}$), 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 didn't, it would violate the supertype's interface.

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

3.3 Dynamic Rules

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

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

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

$$\frac{\hat{e} \longrightarrow \hat{e}' \mid \varepsilon'}{\texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e \longrightarrow \texttt{import}(\varepsilon)\ x = \hat{e}'\ \texttt{in}\ e \mid \varepsilon'}\ \text{(E-Module1)}$$

$$\frac{}{\texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon) \mid \varnothing}\ \text{(E-Module2)}$$

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, EModule1 all reduce a single subexpression.

E-App3 is the standard $\lambda^{\to}$ rule for applying a value to a function, which performs substitution on the function 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 the effect calculus doesn't model the potentially varied return types of functions. Though we haven't defined `unit`, it can be encoded into the existing rules; see Section 4.1.1.

E-Module2 performs module resolution. The (unlabelled) body of code is annotated with the set of effects selected by the module, and then the value being imported is substituted into the body of code.

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

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

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

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 the epsilon calculus is sound. Because the effect-system is orthogonal to the type-system, we must develop an appropriate notion of *effect-soundness*.

Theorem 1 (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 the approximation can only get better as the approximation is successively reduced.

The soundness proof takes the standard approach of showing that progress and preservation hold of the calculus. We begin with a few observations that follow immediately from the typing rules.

Lemma 1 (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 2 (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-MODULE2. □

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 $\hat{e}$. Beyond this observation, the proof is routine.

Lemma 2 (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.

[1] We permit multi-step reductions of length zero to be consistent with Pierce, who defines multi-step reduction as a reflexive relation[5, p. 39].

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 $\hat{\Gamma} \vdash \texttt{import}(\varepsilon)\ x = \hat{v}\ \texttt{in}\ e : \hat{\tau}_A$ `with` ε_A, then we need to be able to type the reduced expression in the same context: $\hat{\Gamma} \vdash [\hat{v}/x]\texttt{annot}(e, \varepsilon) : \hat{\tau}_B$ `with` ε_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:*

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

Then $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon), y : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon)$ `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-MODULE2, 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-MODULE 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), this code is capability-safe.

The last thing we need is to show that $\tau <: \texttt{annot}(\tau, \varepsilon)$. If we know this, then $\tau <: [\hat{v}/x]\texttt{annot}(\tau, \varepsilon)$ by the substitition lemma. The subtyping judgement comes by way of the following pair of lemmas.

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`. The result is obtained by applying the inductive assumptions to the definitions of these judgements. □

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

We now have all we need to show the preservation theorem.

Theorem 3 (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 $\hat{e}_1 : \hat{\tau}_2 \to_\varepsilon \hat{\tau}_3 \texttt{ with } \varepsilon_1$ 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-MODULE. Then $e_A = \texttt{import}(\varepsilon)\ x = \hat{e}\ \texttt{in}\ e$. If the reduction rule used was E-MODULECALL1 then the result follows by applying the inductive hypothesis to $\hat{e}$.

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 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 essentially combines the progress and preservation theorems, and so the proof is a straight-forward application of them. Knowing that single-step reductions are sound, multi-step reductions can straight-forwardly 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 4 (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 5 (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

Our pared down language is nice mathematically, because it is easier to be convinced of properties such as soundness. When writing practical examples it is useful to use higher-level constructs which have been derived from the initial base language. In this section we introduce some of the constructs that we frequently use in examples. Because the core language is sound, any derived extensions are 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 $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to_{\varnothing} \varnothing$. 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.

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

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}} \text{ (T-UNIT)}$$

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

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

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).

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

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

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

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

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

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

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

Figure 4.2: Derived `let` rules.

4.1.3 Tuples

We need tuples to import multiple names.

Chapter 5

Appendix

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 6 (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. □

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` $\varepsilon \cup \varepsilon_1$. □

Bibliography

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

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

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

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

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

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