# A Sample Article for the LIPIcs series[*]

**John Q. Open[1] and Joan R. Access[2]**

1 **Dummy University Computing Laboratory, Address/City, Country**
`open@dummyuniversity.org`

2 **Department of Informatics, Dummy College, Address/City, Country**
`access@dummycollege.org`

**Abstract**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

**1998 ACM Subject Classification** TODO – see `http://www.acm.org/about/class/ccs98-html`

**Keywords and phrases** capability safety, effects, modules

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

## 1 Introduction

The division of software into modules has long been recognised as a technique for achieving code re-use and seperation of concerns [**1968 modular programming conference**]; but not all modules are safe to use. When one part of a system is developed with a particular set of assumptions, the inclusion of modules external to that part can violate these assumptions. There are several reasons why developers may choose to use modules that fundamentally cannot be trusted. A developer may import a library to perform some task, to save the cost of having to re-implement it. The concept of code ownership may come into play [**paper**], whereby different teams and individuals exercise responsibility and maintenance for different sections of code. Each section may come with its own local assumptions and invariants. At a sufficient scale, no one person can be expected to function as an expert over every section of code. The point of interaction between two code sections, owing to mismatched assumptions, is a source of errors. **Is code section the right terminology? Any empirical studies on this?**

Several development heuristics help to achieve safe design, such as the principle of least authority (POLA), which mandates that we limit the components of a system to only the information and resources it needs to perform its task [**?**]. Ensuring that a component satisfies POLA is a difficult task in languages permitting *ambient authority*. In such a language, modules may exercise authority beyond that explicitly declared [**?**].

In Java, the ability to import any public module without restriction is an example of ambient authority. If we wish to track the resources used by class $C_1$, we might start by looking at those resources used directly in the body of $C_1$, or those directly imported by $C_1$. However, if $C_1$ imports $C_2$, and $C_2$ makes use of a resource in its implementation, then $C_1$ will also exercise authority over that resource. Figure 1 demonstrates such an example. As a consequence of ambient authority, one cannot statically determine the set of resources used

[*] This work was partially supported by someone.

by $C_1$ without potentially inspecting every part of the program–a demand at odds with the concept of code ownership.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

class MyList<T> extends ArrayList<T> {
  @Override
  public boolean add(T elem) {
  try {
      Path file = Paths.get(``etc'', ``passwds'');
      Filse.write(file, Charset.forName(``UTF-8''));
  } catch(IOException e) {}
  return super.add(elem);
}
```

```
import MyList;

class Main {
  public static void main(String[] args) {
    MyList<Integer> list = new MyList<>();
    list.add(3);
  }
}
```

**Figure 1** Executing Main.java will result in an attempt at reading the contents of /etc/passwds. Because of ambient authority, we cannot determine that a file system is even at play inspecting every class.

One solution is to only permit the use of resources via *capabilities.* A capability is a unique, unforgeable reference, giving its bearer permission to perform some operation [**?**]. If you possess a capability $C$, you have *authority* over it. In *capability-safe* languages, the only way to use a resource is by using the operations provided to you by some capability $C$ over which you exercise authority. Capability safe languages come with restrictions on how authority over a capability may be gained:

1. There is an initial set of capabilities passed into the program (initial conditions).
2. If a method or function is instantiated by a parent, the parent gains a capability for its child (parenthood).
3. If a method or function 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).

These four principles may be summarised as "no authority amplification" **Miller doesn't use this term**. The only authority in a program must come from these rules, and be derived from previous access; this is known as "only connectivity begets connectivity" [**?**]. These two properties characterise what it means to be capability-safe.

On the other hand, some languages track the use of resources with the notion of an *effect system.* Effects describe intensional information about what happens during the execution of a program [?]. They may be taken to represent imperative commands, stateful updates, or I/O operations. An effect-system extends the notion of a type-system, by ascribing to a piece of code the set of effects that may be incurred when it executes. The type signatures of a module's interface then explicitly states to a developer what effects are at play, allowing them to make an informed decision about whether or not it it safe to use. Some have expressed concerns over the practical use of these effect systems because of their heavyweight nature **CITATION NEEDED**: every piece of code, pure or impure, must be labelled as such. If a refactoring requires module $M$ to make use of a new effect, then every module transitively using $M$ may now also incur the new effect, requiring potentially every part of the system's interface to change. This is again at odds with the notions of code ownership and modularity.

We believe the marriage of effect-systems and capability-safety can provide quick, module-level reasoning, with minimal overhead to developers. Key to this idea is the restriction of "only connectivity begets connectivity", meaning a module $M$ can only use a resource under certain circumstances **ELABORATE, RELATE TO B**. Therefore, we can guarantee the resources used by a module by inspecting the type signatures of those modules it directly imports, without having to inspect the source-code of every module transitively. Then only those modules directly using the privileges granted by a capability are required to declare their effects, reducing the programmer overhead.

Our contribution is to develop an effect-system for a simple, pure, capability-safe object calculus. We develop this idea in two stages. First we describe the epsilon calculus, in which all parts of a program are labelled, as in traditional effect-systems. This requirement is relaxed, and we secondly generalise it to describe programs that may freely mix labelled and unlabelled parts. We show that this type-and-effect system is sound, and give examples to demonstrate that this system is able to map to more complex capability-safe languages, such as Wyvern.

## 2 Capabilities & Modules

**REWORK SECTION**

▶ **Definition 1.** A resource is a capability for some primitive I/O operations.

▶ **Definition 2.** An operation is an invocation of a primitive I/O operation on a resource.

Resources and operations are drawn from fixed sets $R$ and $\Pi$ respectively. They cannot be created during runtime. For simplicity we assume all operations are null-ary; the precise nature of *what* an effect is doing is not of interest to our system, but rather *where* those effects are being used throughout the program.

▶ **Definition 3.** An effect $r.\pi$ is a member of $R \times \Pi$.

▶ **Definition 4.** An effect-set $\varepsilon$ is a subset of $R \times \Pi$.

We usually refer to $(r, \pi)$ as $r.\pi$. For example: `File.append` and `Socket.open` instead of $(\texttt{File}, \texttt{append})$ and $(\texttt{Socket}, \texttt{open})$. For simplicity we assume every operation is defined on every resource, so `Socket.append` would be a valid effect (even though externally you mightn't be able to append to a socket).

Generally, we say a program $e$ has the runtime effect $r.\pi$ if $r.\pi$ is called during the execution of $e$. We say $e$ captuers the runtime effect $r.\pi$ if it has the authority to call $r.\pi$ at some point during execution.

[DISCUSS CAPABILITY-SAFETY: EMILY, E, JOE-E, WYVERN]

[FIRST-CLASS MODULES: SCALA, NEWSPEAK, WYVERN]

## 3 Motivating Examples

When a module requires access to a particular capability we often want to restrict how that capability is used. For example, when we pass a file to a logger, we only expect it to be appending to that file. In a non-effect system, such behaviour cannot be enforced, because access to a capability implies access to its encapsulating operations. Figure 1 shows an example of a logging module which violates this sort of fine-grained restriction on capability use, by writing rather than appending.

```
module Logger
require FileIO

def log(errMsg: String): Unit =
  f.write(errMsg)
```

```
module Main
require FileIO
instantiate Logger(FileIO)
instantiate Client

def main(): Unit =
  Client.run(Logger)
```

```
module Client

type Logger =
  def log(err: String): Unit

def run(logger : Logger) =
  logger.log(‘‘test message’’)
```

**Figure 2** A short program consisting of three modules. Main instantiates the capabilities needed to run the program and passes the Logger to the Client's run method.

From the perspective of the client, it is not immediately clear that its logging capability should have the `File.write` effect. The existence of this behaviour could be for several reasons: it could be a bug on the behalf of a well-intentioned programmer; this particular logging module may be making assumptions different from those of the client; or it may be an intentionally malicious third-patry library. Whatever the case, establishing whether this behaviuor exists would require the client to manually examine the source code of the logger. An effect system can embed this time-consuming process into the type system. Consider the above example, amended to include effect annotations.

An effect-system will now reject this program: `main` declares its effects as being no more than the set {`File.log`}, but its implementation typechecks as having {`File.write`}.

This solution is cumbersome in that it requires every part of the system to be annotated with its effects. If we consider each module as being "owned" by a different team of developers, a local change in the effects produced by one module requires a global refactoring. Maintaining the effect annotations is a considerable overhead; some have cited this heavyweight feel as a point against the practical use of effect systems [**CITATION NEEDED**]. Ideally, developers would only have to annotate the portions of code directly involved in using effects. Modules such as `Main`, which in this case are only involved in the transfer of capabilities,

```
module Logger
require FileIO

def log(errMsg: String): Unit with {File.write} =
  f.write(errMsg)
```

```
module Main
require FileIO
instantiate Logger(FileIO)
instantiate Client

def main(): Unit with {File.log} =
  Client.run(Logger)
```

```
module Client

type Logger =
  def log(err: String): Unit

def run(logger : Logger) with {File.log} =
  logger.log(‘‘test message’’)
```

**Figure 3** The same program now annotates each method with the effects it may invoke, whether directly or indirectly.

would not require annotations.

Because a capability-safe system disallows ambient authority, the only effects possible in `Main` are those in its signature. A summary of the potential effects can be inferred by the type system, leaving manual annotations for the maintainres of the `Client` and `Logging` modules. This is the basic idea underlying our system: we allow the omission of effect annotations in sections of code where the set of effects is not widened.

We shall begin with a formalism for a system where every method is fully-labelled. Although such systems are cumbersome, this process will help introduce the notation and concepts needed to describe the generalised effect-system which allows the mixing of labelled and unlabelled code.

## 4 Epsilon Calculus

The first system is called the epsilon calculus. Its principal unit is the object, and the system is entirely pure. This simplifies reasoning and keeps our system simple and extensible. Importantly, this system is capability-safe: the only effectful capabilities are those passed into the program at the start of the execution, and an object may only gain or access capability by being given them by another object possessing that capability. The objects of the epsilon calculus can also encode the modules of a capability-safe language where these are treated as first-class objects.

## 4.1 Grammar

The epsilon calculus operates on very simple programs $e$, where every declaration $\sigma$ has programmer-annotated labels specifying their effect.

$$
\begin{array}{lcll}
e & ::= & x & \textit{expressions} \\
 & | & r & \\
 & | & \texttt{new}\ x \Rightarrow \overline{\sigma = e} & \\
 & | & e.m(e) & \\
 & | & e.\pi & \\
 & & & \\
\tau & ::= & \{\bar{\sigma}\} & \textit{types} \\
 & | & \{r\} & \\
 & & & \\
\sigma & ::= & \texttt{def}\ m(y:\tau):\tau\ \texttt{with}\ \varepsilon & \textit{decls.}
\end{array}
$$

The fundamental construct is the object, modelled as a record of methods. An object's "this" variable is usually denoted by $x$. Every method declaration $\sigma$ is annotated with an upper-bound $\varepsilon$ on the effects that may happen as a result of executing the method. This upper-bound is not necessarily tight; it may contain effects which don't happen on certain execution paths.

Note that resources and objects are considered distinct, although they work somewhat similarly: methods are invoked on a method, and operations on a resource. A simplifying assumption we make is that every operation takes zero arguments and every method takes one argument. We consider this without loss of generality; in a later section we shall see how multiple-argument methods may be encoded as single-argument methods.

Types in the epsilon calculus are structural. The type with no methods is called `Unit`, and it has a single instance called `unit`. It is used to convey the absence of information (often called `Void` in other languages).

When a method takes `Unit` as its parameter, we shall often omit this information fro brevity. That is, $\texttt{def}\ m(y:\texttt{Unit}:\tau$ and $\texttt{def}\ m():\tau$ are the same declaration.

## 4.2 Static Semantics

A context $\Gamma$ is a mapping from variables to types. In our calculus, $\Gamma$ can prove two sorts of static judgements. The first, $\Gamma \vdash \sigma = e\ \texttt{OK}$ , confirms that the type and effect of $e$ cohere to those of the declaration $\sigma$. The other, $\Gamma \vdash e : \tau\ \texttt{with}\ \varepsilon$, says that $e$ evaluates to something of type $\tau$, and that during execution it will have at most the set $\varepsilon$ as its runtime effects.

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

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

$$\frac{\Gamma, x:\{\bar{\sigma}\} \vdash \overline{\sigma = e}\ \texttt{OK}}{\Gamma \vdash \texttt{new}\ x \Rightarrow \overline{\sigma = e}:\{\bar{\sigma}\}\ \texttt{with}\ \varnothing}\ (\varepsilon\text{-NewObj}) \qquad \frac{\Gamma \vdash e_1:\{r\}\ \texttt{with}\ \varepsilon_1}{\Gamma \vdash e_1.\pi:\texttt{Unit}\ \texttt{with}\ \{r.\pi\} \cup \varepsilon_1}\ (\varepsilon\text{-OperCall})$$

$$\frac{\Gamma \vdash e_1:\{\bar{\sigma}\}\ \texttt{with}\ \varepsilon_1 \quad \Gamma \vdash e_2:\tau_2\ \texttt{with}\ \varepsilon_2 \quad \sigma_i = \texttt{def}\ m_i(y:\tau_2):\tau_3\ \texttt{with}\ \varepsilon_3}{\Gamma \vdash e_1.m_i(e_2):\tau_3\ \texttt{with}\ \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}\ (\varepsilon\text{-MethCall})$$

$\boxed{\Gamma \vdash \sigma = e \ \texttt{OK}}$

$$\frac{\Gamma, y : \tau_2 \vdash e : \tau_3 \ \texttt{with} \ \varepsilon_3 \quad \sigma = \texttt{def} \ m(y : \tau_2) : \tau_3 \ \texttt{with} \ \varepsilon_3}{\Gamma \vdash \sigma = e \ \texttt{OK}} \ (\varepsilon\text{-ValidImpl})$$

In the rules $\varepsilon$-Var, $\varepsilon$-Resource, and $\varepsilon$-NewObj, the result types with no effect. Although these forms may nest capabilities for some resource, it is not an effect nuless something is done with them, such as invoking a method or an operation; these situations are, respectively, the rules and $\varepsilon$-MethCall and $\varepsilon$-OperCall.

Our rules make some simplifying assumptions. In practice, some effects allow a program to obtain data from an external source during runtime. An example of such an effect might be `File.read` or `Socket.read`. Because our motivating examples do not require such effects, we will assume $r.\pi$ always returns `unit`. Lastly, this system has no subtyping rules. Although this makes programs less tractable, for expository purposes we will not need them yet. Later on the epsilon* calculus will introduce subtyping rules.

## 4.3 Dynamic Semantics

The notation for a single-step reduction is $e \longrightarrow e' \mid \varepsilon$. This should be read as $e$ being reduced to $e'$, during which all the effects in $\varepsilon$ take place. For now, because it is a single small-step reduction, $\varepsilon$ will be either a singleton or empty. It should be noted that the $\varepsilon$ in $e \longrightarrow e' \mid \varepsilon$ is the *true* set of runtime effects. Compare this with a static judgement like $\Gamma \vdash e : \tau \ \texttt{with} \ \varepsilon$, where $\varepsilon$ is a conservative approximation to the runtime effects.

For a program to perform operation calls it must know what resources and operations are available to use. These are enumerated in a resource context $\langle R, \Pi \rangle$. These lists are fixed throughout the runtime. A valid operation is a pair $r.\pi \in R \times \Pi$. We use the notation $\langle R, \Pi \rangle \vdash e \longrightarrow e' \mid \varepsilon$ to mean the reduction on the right-hand side is true in a particular resource context. If $e$ is closed under $\Gamma$, then $\langle R, \Pi \rangle$ is a *corresponding runtime* if every operation call $\pi$ appearing in $e$ is in $\Pi$ and every resource $r$ appearing in $e$ is in $R$.

If a non-variable expression cannot be reduced under the dynamic semantics it is called a *value*. If we know an expression is a value then we usually denote it by $v$ instead of $e$.

$$\begin{array}{rcll} v & ::= & r & \textit{values} \\ & \mid & \texttt{new} \ x \Rightarrow \overline{\sigma = e} & \end{array}$$

Our notation for variable substitution is $[v/z]e$. This yields an expression with the same structure as $e$, where every free occurrence of $z$ has been replaced by the value $v$. Substitution on non-values is undefined. A precise definition is given in appendix **A.1.** The short-hand $[v_1/y_1, v_2/y_2]e$ is the same as $[v_2/y]([v_1/x]e)$. Note the order of the variables has been. This means the substitutions occur left-to-right, as they appear in the short-hand.

We are ready to provide the dynamic semantics for the epsilon calculus. It is given in a small-step style.

$\boxed{\langle R, \Pi \rangle \vdash e \longrightarrow e \mid \varepsilon}$

$$\frac{\langle R, \Pi\rangle \vdash e_1 \longrightarrow e_1' \mid \varepsilon}{\langle R, \Pi\rangle \vdash e_1.m(e_2) \longrightarrow e_1'.m(e_2) \mid \varepsilon} \text{ (E-M}\textsc{eth}\text{C}\textsc{all}\text{1)}$$

$$\frac{\langle R, \Pi\rangle \vdash v_1 = \texttt{new}\ x \Rightarrow \overline{\sigma = e} \quad e_2 \longrightarrow e_2' \mid \varepsilon}{\langle R, \Pi\rangle \vdash v_1.m(e_2) \longrightarrow v_1.m(e_2') \mid \varepsilon} \text{ (E-M}\textsc{eth}\text{C}\textsc{all}\text{2)}$$

$$\frac{\langle R, \Pi\rangle \vdash v_1 = \texttt{new}\ x \Rightarrow \overline{\sigma = e} \quad \texttt{def}\ m(y : \tau_1) : \tau_2\ \texttt{with}\ \varepsilon = \varepsilon \in \overline{\sigma = e}}{\langle R, \Pi\rangle \vdash v_1.m(v_2) \longrightarrow [v_1/x, v_2/y]e \mid \varnothing} \text{ (E-M}\textsc{eth}\text{C}\textsc{all}\text{3)}$$

$$\frac{\langle R, \Pi\rangle \vdash e_1 \longrightarrow e_1' \mid \varepsilon}{\langle R, \Pi\rangle \vdash e_1.\pi \longrightarrow e_1'.\pi \mid \varepsilon} \text{ (E-O}\textsc{per}\text{C}\textsc{all}\text{1)} \qquad \frac{r \in R \quad \pi \in \Pi}{\langle R, \Pi\rangle \vdash r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}} \text{ (E-O}\textsc{per}\text{C}\textsc{all}\text{2)}$$

Since programs contain no mutable state, we perform method execution by substituting the formal parameters of a method body for the actual values supplied to it by a method call. This is what happens in E-METHCALL3. Furthermore, because E-METHCALL2 requires the receiver to have been reduced to a value, this enforces a left-to-right evaluation order.

When an operation is performed on a resource literal it is reduced to `unit`, per our convention. This is the only case in which a runtime effect occurs.

We are only interested in reduction of programs which are well-formed according to a particular typing context $\Gamma$. $\Gamma$ can prove that $e$ is well-formed if every name occurring free in $e$ is defined in $\Gamma$.

▶ **Definition 5** (Closed Expression). $e$ is *closed* under $\Gamma$ if $z \in \texttt{freevars}(e) \implies z \in \Gamma$.

A precise definition may be found in the appendix. Closedness of programs is invariant under reduction. From this point onwards we only consider the reduction of programs which are well-typed in some $\Gamma$, and implicitly assume they are also closed under $\Gamma$.

▶ **Lemma 6** (Closedness). *Suppose $e_A$ is closed under $\Gamma$ and $\langle R, \Pi\rangle$ is a corresponding runtime. If $\langle R, \Pi\rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ then $e_B$ is closed under $\Gamma$.*

**Proof.** By induction on $\langle R, \Pi\rangle \vdash e \longrightarrow e' \mid \varepsilon$. ◀

Using single-step reductions we can define multi-step reduction. The notation is $\langle R, \Pi\rangle \vdash e \longrightarrow^* e' \mid \varepsilon$, which means $e$ can be reduced to $e'$ via *zero* or more transitive applications of the single-step reductions; their cumulative effects are the set $\varepsilon$.

$\boxed{\langle R, \Pi\rangle \vdash e \longrightarrow^* e \mid \varepsilon}$

$$\frac{}{\langle R, \Pi\rangle \vdash e \longrightarrow^* e \mid \varnothing} \text{ (E-M}\textsc{ulti}\text{S}\textsc{tep}\text{1)} \qquad \frac{\langle R, \Pi\rangle \vdash e \longrightarrow e' \mid \varepsilon}{\langle R, \Pi\rangle \vdash e \longrightarrow^* e' \mid \varepsilon} \text{ (E-M}\textsc{ulti}\text{S}\textsc{tep}\text{2)}$$

$$\frac{\langle R, \Pi\rangle \vdash e \longrightarrow^* e' \mid \varepsilon_1 \quad \langle R, \Pi\rangle \vdash e' \longrightarrow^* e'' \mid \varepsilon_2}{\langle R, \Pi\rangle \vdash e \longrightarrow^* e'' \mid \varepsilon_1 \cup \varepsilon_2} \text{ (E-M}\textsc{ulti}\text{S}\textsc{tep}\text{3)}$$

## 4.4 Soundness of Epsilon Calculus

In proving soundness we go down the traditional route of proving progress and a form of preservation which guarantees that types and effects both are preserved under reduction. Our first lemma is canonical forms, which follows by inversion on the typing rules.

▶ **Lemma 7** (Canonical Forms). *If $v$ is a value then the following are true.*
1. *If $\Gamma \vdash v : \{r\}$ `with` $\varepsilon$, then $v = r$ for a resource $r \in \Gamma$ and $\varepsilon = \varnothing$.*
2. *If $\Gamma \vdash v : \{\bar{\sigma}\}$ `with` $\varepsilon$, then $v =$ `new` $x \Rightarrow \overline{\sigma = e}$ and $\varepsilon = \varnothing$.*

The most useful result here is that a value necessarily types as having no effect. From canonical forms we can prove the progress theorem.

▶ **Theorem 8** (Progress). *If $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$, either $e_A$ is a value or a single-step $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ can be applied in a corresponding runtime.*

**Proof.** By induction on $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ and then by considering which subexpressions are values. If $e_A$ is a value it holds immediately, otherwise $e_A$ is a method call or an operation call. In either case, if it contains a reducible subexpression, we may apply a reduction to that subexpression. If we assume it contains no reducible subexpressions, then we are left with two cases.

Case 1: $e_A = v_1.\pi$. E-OperCall2 gives the reduction $\langle R, \Pi \rangle \vdash v_1.\pi \longrightarrow$ `unit` $\mid \{r.\pi\}$.

Case 2: $e_A = v_1.m_i(v_2)$. As this expression types, $v_1$ contains a declaration $\sigma_i = e_i$ for method $m_i$. Then E-MethCall3 gives the reduction $\langle R, \Pi \rangle \vdash v_1.m_i(v_2) \longrightarrow e_i \mid \varnothing$. ◀

To prove preservation we need the type of an expression to be preserved under reduction. Since the epsilon calculus has no subtyping, this means the type before reduction will be the same as the type after reduction.

We also need the preservation of static-effects under reduction.

If $\Gamma \vdash e_A : \tau$ `with` $\varepsilon_A$ and $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ and $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$, we say the static-effects are preserved if $\varepsilon_A = \varepsilon \cup \varepsilon_B$. If we interpret $\varepsilon$ as the effects which happened during the small-step, and $\varepsilon_B$ as the effects which have yet to happen, then together these form the effects $\varepsilon_A$, which are those which were yet to happen before the reduction. Our last lemma states that an expression's type and static-effects are preserved when substituting a variable for an appropriate value.

▶ **Lemma 9** (Substitution Lemma). *If $v$ is a value and $\Gamma, y : \tau' \vdash e : \tau$ `with` $\varepsilon$ and $\Gamma \vdash v : \tau'$ `with` $\varnothing$ then $\Gamma \vdash [v/z]e : \tau$ `with` $\varepsilon$.*

**Proof.** By induction on $\Gamma, z : \tau' \vdash e : \tau$ `with` $\varepsilon$. ◀

▶ **Theorem 10** (Preservation Theorem). *If $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ and $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ in a corresponding runtime, then the following are true:*

- $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$
- $\tau_B = \tau_A$
- $\varepsilon_A = \varepsilon_B \cup \varepsilon$

**Proof.** By induction on $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ and then on $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$. If $e_A$ is a value the result holds vacuously. Otherwise $e_A$ is a non-value, and therea re two typing rules to consider.

The first rule is $\varepsilon$-OperCall. Then $e_A = e_1.\pi$. If the reduction used was $\langle R, \Pi \rangle \vdash e_1.\pi \longrightarrow e_1'.\pi \mid \varepsilon$, then the result follows by applying the induction hypothesis to the subreduction. Otherwise the reduction was $\langle R, \Pi \rangle \vdash r.\pi \longrightarrow \texttt{unit} \mid \varnothing$, in which case the reduction follows by using $\varepsilon$-NewObj to type $\Gamma \vdash \texttt{unit} : \texttt{Unit with } \varnothing$.

The second rule is $\varepsilon$-MethCall. Then $e_A = e_1.m_i(e_2)$. If the reduction used involved reducing a subexpression, the result follows by induction on the appropriate subreduction. Otherwise the reduction used was $\langle R, \Pi \rangle \vdash v_1.m_i(v_2) \longrightarrow [v_1/x, v_2/y]e_i \mid \varnothing$, where $e_i$ is the method body for $m_i$. The result follows by inversion on $\Gamma \vdash e_1 : \{\bar{\sigma}\}$ `with` $\varnothing$ to obtain a typing judgement for $e_i$, and then two applications of the substitution lemma to $[v_1/x, v_2/y]e_i$. ◀

Putting together preservation and progress immediately yields soundness for single-step reductions. To obtain soundness for multi-step reductions, we induct on the length of a multi-step erduction and appeal to the soundness of single-step reductions.

▶ **Theorem 11** (Single-Step Soundness). *If* $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ *and then either* $e_A$ *is a value or the following are true in a corresponding runtime:*
- $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon_B$
- $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$
- $\tau_A = \tau_B$
- $\varepsilon_A = \varepsilon \cup \varepsilon_B$

▶ **Theorem 12** (Multi-Step Soundness). *If* $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ *and then either* $e_A$ *is a value or the following are true in a corresponding runtime:*
- $\langle R, \Pi \rangle \vdash e_A \longrightarrow^* e_B \mid \varepsilon_B$
- $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$
- $\tau_A = \tau_B$
- $\varepsilon_A = \varepsilon \cup \varepsilon_B$

## 4.5 Desugaring Rules

The epsilon calculus is able to map to pure, capability-safe object-oriented languages with first-class modules. In this section we show how to encode common language constructs.

A let expression has the form `let` $z = e_A$ `in` $e_B$. If $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ and $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$, the let can be desugared as in figure 3.

```
let x = e₁ in e₂
```

```
new x ⇒ {
   def doIt(y: τ_A): τ_B with ε_B = e_B
}.doIt(e_A)
```

**Figure 4** Let expression desugaring.

A basic program in one file may be considered as consisting of a single module `MainModule` with a method `main`. The body of `main` will contain the expression to be executed in the file, which the desugared program invokes. Any resources used must be defined in the runtime $\langle R, \Pi \rangle$ in which execution takes place.
Where there are multiple modules in a program we must instantiate them before invoking main. This requires resolving module dependencies. In our programs there are two sorts of (impure) capabilities that a module might require.

```
FileIO.write
```

```
let MainModule = new x ⇒ {
   def main(): Unit = FileIO.write
} in MainModule.main()
```

**Figure 5** A basic program desugared into its implicit “main module” form. An appropriate runtime would be $\langle FileIO, write\rangle$

The first are those drawn from the fixed set of resources $R$. These are assumed to be supplied by some executing environment For example, `FileIO` could be a file-handle passed to the program via command-line. We assume these capabilities exist in the runtime $\langle R, \Pi\rangle$.

The other sort of impure capabilities are those objects which use a resource, directly or indirectly. If `Module` depends on $C_1$ and $C_2$, we must instantiate modules $C_1$ and $C_2$ first and then pass them to module $A$ as arguments. If we assume $C_1$ and $C_2$ have already been created, `Module` can be created by nesting several objects inside each other. Each successive object will have a single method which binds one of the required modules. The inner-most object will have a single method returning an object containing all the methods defined by `Module`. Figure 5 demonstrates how this works.

```
module Module
require Capability1 as c₁
require Capability2 as c₂

def method₁(): Unit = Capability₁.action()
def method₂(): Unit = Capability₂.action()
```

```
let c₁ = .... in
let c₂ = .... in

let Module = new x ⇒ {
   def resolveC₁(capability₁: Capability₁Type) with ∅ =
      new x ⇒ {
         def resolveC₂(capability₂: Capability₂Type) with ∅ =
            new x ⇒ {
               def method₁(): Unit =
                  capability₁.action()
               def method₂(): Unit =
                  capability₂.action()
         }
      }
}.resolveC₁(c₁).resolveC₂(c₂)
```

**Figure 6** Desugaring of the initialisation of `Module`, which has dependencies `C1` and `C2`.

We are now able to desugar the example from section 2. This example can be written in the epsilon calculus, as in figure 6. It will not typecheck, highlighting the mismatch between specification and implementation. This is because in order to typecheck `loggerModule` using $\varepsilon$-NewObj, we must first typecheck the implementation of `loggerModule.log` against its declaration using $\varepsilon$-ValidImpl. This rule cannot be appiled, as $\texttt{FileIO} : \{\texttt{FileIO}\} \vdash$

FileIO.write : unit with {FileIO.write}, and the effect-set {FileIO.write} is different from {FileIO.append}.

```
type Logger is
   def log(): Unit with {FileIO.append}
in

let loggerModule = new x ⇒ {
   def log(): Unit with {FileIO.append} =
      FileIO.write
} in

let clientModule = new x ⇒ {
   def run(logger: Logger): Unit with {FileIO.append} =
      logger.log()
} in

let mainModule = new x ⇒ {
   def main(): Unit with {FileIO.append} =
      clientModule.run(loggerModule)
}

in mainModule.main()
```

**Figure 7** An effect-annotated version of the logger example wil fail to typecheck, highlighting the discrepancy between the interface of `loggerModule.log` and its implementation.

# A Proofs & Definitions For Epsilon Calculus

## A.1 Well-Formedness

▶ **Definition 13** (freevars).

- $\texttt{freevars}(r) = \varnothing$
- $\texttt{freevars}(x) = \varnothing$
- $\texttt{freevars}(e_1.m(e_2)) = \texttt{freevars}(e_1) \cup \texttt{freevars}(e_2)$
- $\texttt{freevars}(e_1.\pi) = \texttt{freevars}(e_1)$
- $\texttt{freevars}(\texttt{new}\ x \Rightarrow \overline{\sigma = e}) = \bigcup(\texttt{freevars}(e_i) \setminus \{y, x\})$, where $\texttt{def}\ m_i(y : \tau_2) : \tau_3\ \texttt{with}\ \varepsilon$

▶ **Lemma 14** (Closedness). *Suppose $e_A$ is closed under $\Gamma$ and $\langle R, \Pi \rangle$ is a corresponding runtime. If $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ then $e_B$ is closed under $\Gamma$.*

**Proof.** By induction on $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$.

$\boxed{\text{Case: } \varepsilon\text{-MethCall1}}$ By induction $\texttt{freevars}(e'_1) \subseteq \Gamma$, so $\texttt{freevars}(e'_1.m(e_2)) \subseteq \Gamma$.

$\boxed{\text{Case: } \varepsilon\text{-MethCall2}}$ By induction $\texttt{freevars}(e'_2) \subseteq \Gamma$, so $\texttt{freevars}(e_1.m(e'_2)) \subseteq \Gamma$.

$\boxed{\text{Case: } \varepsilon\text{-MethCall3}}$ Then $\langle R, \Pi \rangle \vdash v_1.m_i(v_2) \longrightarrow [v_1/x, v_2/y]e_1$. Because $e_A$ is closed under $\Gamma$ so is its subexpression $v_1$, and so $\texttt{freevars}(\texttt{e}_\texttt{i}) \setminus \texttt{y}, \texttt{x} \subseteq \Gamma$. Then $\texttt{freevars}([v_1/x, v_2/y]e_i) = \texttt{freevars}(e_i) \setminus \{x, y\}$, so $[v_1/x, v_2/y]e_i$ is also closed under $\Gamma$.

$\boxed{\text{Case: } \varepsilon\text{-OperCall1}}$ By induction $e'_1$ is closed under $\Gamma$. Then $e'_1.\pi$ is too.

$\boxed{\text{Case: } \varepsilon\text{-OperCall2}}$ Then $e_B = \texttt{unit}$ and $\texttt{freevars}(\texttt{unit}) = \varnothing \subseteq \Gamma$ trivially.

◀

## A.2 Definition of Substitution

Informally, $[e'/z]e$ produces an expression where every free occurrence of $z$ has been replaced by $e'$. We use the convention of $\alpha$-conversion, which says that free variables may be renamed to avoid accidental capture. [ cite TAPL page 71]

$$
\begin{aligned}
&[e'/z]z = e'\\
&[e'/z]y = y, \text{ if } y \neq z\\
&[e'/z]r = r\\
&[e'/z](e_1.m(e_2)) = ([e'/z]e_1).m([e'/z]e_2)\\
&[e'/z](e_1.\pi) = ([e'/z]e_1).\pi\\
&[e'/z](\texttt{new}\ x \Rightarrow \overline{\sigma = e}) = \texttt{new}\ x \Rightarrow \overline{\sigma = [e'/z]e}, \text{ if } z \neq x \text{ and } z \notin \texttt{freevars}(e_i)
\end{aligned}
$$

## A.3 Canonical Forms

▶ **Lemma 15** (Canonical Forms). *If $v$ is a value then the following are true.*

1. *If $\Gamma \vdash v : \{r\}\ \texttt{with}\ \varepsilon$, then $v = r$ for a resource $r \in \Gamma$ and $\varepsilon = \varnothing$.*
2. *If $\Gamma \vdash v : \{\bar{\sigma}\}\ \texttt{with}\ \varepsilon$, then $v = \texttt{new}\ x \Rightarrow \overline{\sigma = e}$ and $\varepsilon = \varnothing$.*

**Proof.** The first two judgements appear exactly once in the conclusions of $\varepsilon$-Resource and $\varepsilon$-NewObj respectively, yielding **1** and **2** directly. The only other rule for typing values is $\varepsilon$-Var, and in the conclusions of all three rules, $\varepsilon = \varnothing$. ◀

## A.4 Progress Theorem

▶ **Theorem 16** (Progress). *If* $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$*, either* $e_A$ *is a value or a single-step* $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ *can be applied in a corresponding runtime.*

**Proof.** By induction on the derivation of $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$. If $e_A$ is not a value then one of two rules could have been used.

$\boxed{\text{Case: } \varepsilon\text{-MethCall}}$ Then $e_A = e_1.m_i(e_2)$ and the following are known.

1. $e_A : \tau_3$ `with` $\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$
2. $e_1 : \{\bar{\sigma}\}$ `with` $\varepsilon_1$
3. $e_2 : \tau_2$ `with` $\varepsilon_2$
4. $\sigma_i =$ `def` $m_i(y : \tau_2) : \tau_3$ `with` $\varepsilon_3$

We look at the cases for when $e_1$ and $e_2$ are values. If $e_1$ is not a value then by the inductive hypothesis applied to **2** there is a reduction $\langle R, \Pi \rangle \vdash e_1 \longrightarrow e_1' \mid \varepsilon$. Then E-MethCall1 gives the reduction $\langle R, \Pi \rangle \vdash e_1.m_i(e_2) \longrightarrow e_1'.m_i(e_2) \mid \varepsilon$.

If $e_2$ is not a value then by the inductive hypothesis applied to **3** there is a reduction $\langle R, \Pi \rangle \vdash e_2 \longrightarrow e_2' \mid \varepsilon$. Without loss of generality, $e_1$ is a value. Then E-MethCall2 gives the reduction $\langle R, \Pi \rangle \vdash e_1.m_i(e_2) \longrightarrow e_1.m_i(e_2') \mid \varepsilon$.

Otherwise $e_1$ and $e_2$ are values. By canonical forms, $e_1 =$ `new` $x \Rightarrow \overline{\sigma = e}$. By inversion on **2**, there is a declaration `def` $m_i(y : \tau_2) : \tau_3$ `with` $\varepsilon_3 = e_i \in \overline{\sigma = e}$. Then E-MethCall3 gives the reduction $\langle R, \Pi \rangle \vdash e_1.m_i(e_2) \longrightarrow [e_1/x, e_2/y]e_i \mid \varnothing$.

◀

## A.5 Substitution Lemma

▶ **Lemma 17** (Substitution Lemma). *If* $v$ *is a value and* $\Gamma \vdash v : \tau'$ `with` $\varnothing$ *and* $\Gamma, z : \tau' \vdash e : \tau$ `with` $\varepsilon$*, then* $\Gamma \vdash e : [v/z]e : \tau$ `with` $\varepsilon$.

**Proof.** By induction on $\Gamma, z : \tau' \vdash e : \tau$ `with` $\varepsilon$. If $z$ does not occur free in $e$, then $[e'/z]e = e$ and the theorem holds trivially. Therefore in each case we assume $z$ occurs free in $e$.

$\boxed{\text{Case: } \varepsilon\text{-Var}}$ Then $\Gamma, z : \tau' \vdash z : \tau$ `with` $\varnothing$ by canonical forms. After substitution, $[v/z]z = v$, so $\Gamma \vdash [v/z]z : \tau$ `with` $\varepsilon$.

$\boxed{\text{Case: } \varepsilon\text{-Resource}}$ Since $[e'/z]r = r$ the statement holds vacuously.

$\boxed{\text{Case: } \varepsilon\text{-OperCall}}$ Then $e = e_1.\pi$. By inversion on $\varepsilon$-OperCall, we learn $\Gamma, z : \tau' \vdash e_1 : \{r\}$ `with` $\varepsilon_1$, where $\varepsilon = \varepsilon_1 \cup \{r.\pi\}$. By inductive assumption, $\Gamma \vdash [v/z]e_1 : \{r\}$ `with` $\varepsilon_1$. By definition, $[v/z](e_1.\pi) = ([v/z]e_1).\pi$. By application of $\varepsilon$-OperCall we learn $\Gamma \vdash [v/z](e_1.\pi) : \{r\}$ `with` $\varepsilon_1 \cup \{r.\pi\}$.

$\boxed{\text{Case: } \varepsilon\text{-MethCall}}$ Then $e = e_1.m_i(e_2)$. From inversion on $\varepsilon$-MethCall we learn:

1. $\sigma_i =$ `def` $m_i(y : \tau_2) : \tau_3$ `with` $\varepsilon_3$
2. $\Gamma, z : \tau' \vdash e_1 : \tau_1$ `with` $\varepsilon_1$
3. $\Gamma, z : \tau' \vdash e_2 : \tau_2$ `with` $\varepsilon_2$
4. $\tau = \tau_3$ and $\varepsilon = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$

By inductive assumption on **2** and **3** we get a pair of judgements $\Gamma \vdash [v/z]e_1 : \tau_1 \texttt{ with } \varepsilon_1$ and $\Gamma \vdash [v/z]e_2 : \tau_2 \texttt{ with } \varepsilon_2$. By definition, $[v/z](e_1.m_i(e_2)) = ([v/z]e_1).m_i([v/z]e_2)$. From the two judgements we just derived and simplification by **4**, we can apply $\varepsilon$-MethCall to obtain $\Gamma \vdash [v/z](e_1.m_i(e_2)) : \tau_3 \texttt{ with } \varepsilon$.

$\boxed{\text{Case: } \varepsilon\text{-NewObj}}$ Then $e = \texttt{new } x \Rightarrow \overline{\sigma = e}$ and $z$. Consider some method $\sigma_i = e_i$. From canonical forms, the assumption from the theorem statement can be more specifically written as $\Gamma, z : \tau' \vdash e : \{\bar{\sigma}\} \texttt{ with } \varnothing$. By inversion on this judgement we know $\Gamma, z : \tau', x : \{\bar{\sigma}\} \vdash \overline{\sigma = e} \texttt{ OK}$. The only rule with this conclusion is $\varepsilon$-ValidImpl. By inversion on that rule we learn:

1. $\sigma_i = \texttt{def } m_i(y : \tau_1) : \tau_2 \texttt{ with } \varepsilon$
2. $\Gamma, z : \tau', x : \{\bar{\sigma}\}, y : \tau_1 \vdash e_i : \tau_2 \texttt{ with } \varepsilon$

By inductive assumption applied to **2** we get $\Gamma, x : \{\bar{\sigma}\}, y : \tau_1 \vdash [v/z]e_i : \tau_2 \texttt{ with } \varepsilon$. Then by applying $\varepsilon$-ValidImpl we get $\Gamma, x : \{\bar{\sigma}\} \vdash \sigma_i = [v/z]e_i \texttt{ OK}$. The method $\sigma_i = e_i$ was arbitrary, so $\Gamma, x : \{\bar{\sigma}\} \vdash \overline{\sigma = e} \texttt{ OK}$. Then by applying $\varepsilon$-NewObj we get $\Gamma \vdash [v/z]e : \{\bar{\sigma}\} \texttt{ with } \varnothing$. ◀

## A.6 Preservation Theorem

▶ **Theorem 18** (Preservation Theorem). *If* $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ *and* $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$ *in a corresponding runtime, then the following are true:*

- $\Gamma \vdash e_B : \tau_B \texttt{ with } \varepsilon_B$
- $\tau_B = \tau_A$
- $\varepsilon_A = \varepsilon_B \cup \varepsilon$

**Proof.** By induction on $\Gamma \vdash e_A : \tau_A \texttt{ with } \varepsilon_A$ and then on $\langle R, \Pi \rangle \vdash e_A \longrightarrow e_B \mid \varepsilon$. If $e_A$ is a value no reduction can be applied so the theorem statement vacuously holds. Otherwise there are two rules to consider.

$\boxed{\text{Case: } \varepsilon\text{-OperCall}}$ Then $e_A = e_1.\pi$ and we know:

1. $\Gamma \vdash e_A : \texttt{Unit with } \{r.\pi\} \cup \varepsilon_1$
2. $e_1 : \{r\} \texttt{ with } \varepsilon_1$

If the reduction rule used was E-OperCall1 then $\langle R, \Pi \rangle \vdash e_1 \longrightarrow e_1' \mid \varepsilon$ and $\langle R, \Pi \rangle \vdash e_1.\pi \longrightarrow e_1'.\pi \mid \varepsilon$. By inductive assumption, $\Gamma \vdash e_1' : \{r\} \texttt{ with } \varepsilon_1'$, where $\varepsilon_1 = \varepsilon \cup \varepsilon_1'$. By application of $\varepsilon$-OperCall we have $\Gamma \vdash e_1'.\pi : \texttt{Unit with } \{r.\pi\} \cup \varepsilon_1'$. Then $\varepsilon_B \cup \varepsilon = \{r.\pi\} \cup \varepsilon_1' \cup \varepsilon = \{r.\pi\} \cup \varepsilon_1 = \varepsilon_A$.

If the rule used was E-OperCall2 then $\langle R, \Pi \rangle \vdash r.\pi \longrightarrow \texttt{unit} \mid \{r.\pi\}$. By canonical forms, $\varepsilon_1 = \varnothing$, so by **1** we have $\varepsilon_A = \{r.\pi\}$. By a degenerate case of $\varepsilon$-NewObj, $\Gamma \vdash \texttt{unit} : \texttt{Unit with } \varnothing$. Then $\varepsilon_B \cup \varepsilon = \varnothing \cup \{r.\pi\} = \varepsilon_A$.

$\boxed{\text{Case: } \varepsilon\text{-MethCall}}$ Then $e_A = e_1.m_i(e_2)$ and the following are true.

1. $\Gamma \vdash e_A : \tau_3 \texttt{ with } \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$
2. $\Gamma \vdash e_1 : \{\bar{\sigma}\} \texttt{ with } \varepsilon_1$
3. $\Gamma \vdash e_2 : \tau_2 \texttt{ with } \varepsilon_2$
4. $\sigma_i = \texttt{def } m_i(y : \tau_2) : \tau_3 \texttt{ with } \varepsilon_3$

If the reduction rule used was E-MethCall1 then $\langle R, \Pi\rangle \vdash e_1 \longrightarrow e_1' \mid \varepsilon$ and $\langle R, \Pi\rangle \vdash e_1.m_i(e_2) \longrightarrow e_1'.m_i(e_2) \mid \varepsilon$. By inductive assumption, $\Gamma \vdash e_1' : \{\bar{\sigma}\}$ `with` $\varepsilon_1'$, where $\varepsilon_1 = \varepsilon_1' \cup \varepsilon$. By applying $\varepsilon$-MethCall we have $\Gamma \vdash e_1'.m_i(e_2) : \tau_3$ `with` $\varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_3$. Then $\varepsilon_B \cup \varepsilon = \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_3 \cup \varepsilon = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 = \varepsilon_A$.

If the reduction rule used was E-MethCall2 then we know $\langle R, \Pi\rangle \vdash e_2 \longrightarrow e_2' \mid \varepsilon$ and $e_1.m_i(e_2) \longrightarrow e_1.m_i(e_2') \mid \varepsilon$. $\Gamma \vdash e_1.m_i(e_2') : \tau_2$ `with` $\varepsilon_1 \cup \varepsilon_2' \cup \varepsilon_3$. By inductive assumption, $\Gamma \vdash e_2' : \tau_2$ `with` $\varepsilon_2'$, where $\varepsilon_2 = \varepsilon \cup \varepsilon_2'$. By applying $\varepsilon$-MethCall we have $\Gamma \vdash e_1.m_i(e_2) : \tau_3$ `with` $\varepsilon_1 \cup \varepsilon_2' \cup \varepsilon_3$. Then $\varepsilon \cup \varepsilon_B = \varepsilon \cup \varepsilon_1 \cup \varepsilon_2' \cup \varepsilon_3 = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 = \varepsilon_A$. Additionally, $\varepsilon_1 = \varnothing$ by canonical forms.

If the reduction rule used was E-MethCall3 then we know $e_1 = v_1$ and $e_2 = v_2$ are values and $\langle R, \Pi\rangle \vdash v_1.m_i(v_2) \longrightarrow [v_1/x, v_2/y]e_i \mid \varnothing$. By inversion on **2** there is a typing judgement $\Gamma, x : \{\bar{\sigma}\}, y : \tau_2 \vdash e_i : \tau_3$ `with` $\varepsilon_3$. By two applications of the substitution lemma, $\Gamma \vdash [v_1/x, v_2/y]e_i : \tau_3$ `with` $\varepsilon_3$. By canonical forms, $\varepsilon_1 = \varepsilon_2 = \varnothing$. Then $\varepsilon_B \cup \varepsilon = \varepsilon_3 \cup \varnothing = \varepsilon_A$. ◀

## A.7 Soundness Theorems

▶ **Theorem 19** (Single-Step Soundness). *If $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ and then either $e_A$ is a value or the following are true in a corresponding runtime:*

- $\langle R, \Pi\rangle \vdash e_A \longrightarrow e_B \mid \varepsilon_B$
- $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$
- $\tau_A = \tau_B$
- $\varepsilon_A = \varepsilon \cup \varepsilon_B$

**Proof.** If $e_A$ is not a value then a reduction $\langle R, \Pi\rangle \vdash e_A \longrightarrow e_B \mid \varepsilon_B$ exists by the progress theorem. Type-and-effect soundness holds by the preservation theorem. ◀

▶ **Theorem 20** (Multi-Step Soundness). *If $\Gamma \vdash e_A : \tau_A$ `with` $\varepsilon_A$ and then either $e_A$ is a value or the following are true in a corresponding runtime:*

- $\langle R, \Pi\rangle \vdash e_A \longrightarrow^* e_B \mid \varepsilon_B$
- $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$
- $\tau_A = \tau_B$
- $\varepsilon_A = \varepsilon \cup \varepsilon_B$

**Proof.** By induction on the number of small-step reductions in $\langle R, \Pi\rangle \vdash e_A \longrightarrow^* e_B \mid \varepsilon$. If there are no small-steps then $e_B \cup \varepsilon = \varepsilon_B = e_A$ and the result holds trivially. If there is one small-step then the result holds by small-step soundness.

Otherwise there are $n > 1$ small-steps then consider a multi-step reduction $\langle R, \Pi\rangle \vdash e_A \longrightarrow e_C \mid \varepsilon$ of length $n$. This consists of a multi-step reduction of length $n$, $\langle R, \Pi\rangle \vdash e_A \longrightarrow^* e_B \mid \varepsilon_1$, and a single-step reduction $\langle R, \Pi\rangle \vdash e_B \longrightarrow e_C \mid \varepsilon_2$, where $\varepsilon = \varepsilon_1 \cup \varepsilon_2$. By inductive assumption, $\Gamma \vdash e_B : \tau_B$ `with` $\varepsilon_B$, where $\varepsilon_1 \cup \varepsilon_B = \varepsilon_A$, and $\tau_B = \tau_A$. By small-step soundness, $\Gamma \vdash e_C : \tau_C$ `with` $\varepsilon_C$, where $\varepsilon_B = \varepsilon_C \cup \varepsilon_2$ and $\tau_B = \tau_C$. By transitivity, $\tau_A = \tau_C$. By comparing equalities, $\varepsilon_A = \varepsilon_1 \cup \varepsilon_B = \varepsilon_1 \cup \varepsilon_C \cup \varepsilon_2 = \varepsilon \cup \varepsilon_C$. ◀

## B Bibliography