Capabilities: Effects for Free

ANONYMOUS AUTHOR(S)

Capabilities are increasingly used to reason informally about the properties of secure systems. Can capabilities also aid in *formal* reasoning? To answer this question, we examine a calculus that uses effects to capture resource use and extend it with a rule that captures the essence of capability-based reasoning. We demonstrate that capabilities provide a way to reason for free about effects: we can bound the effects of an expression based on the capabilities to which it has access. This reasoning is “free” in that they rely only on typechecking (not effect-checking); they do not require the programmer to add effect annotations within the expression, nor do they require a tool to analyse the expression’s structure for its effects. Our result sheds light on the essence of what capabilities provide and suggests useful ways of integrating lightweight capability-based reasoning into languages.

CCS Concepts: •**Software and its engineering** → *Semantics;*

ACM Reference format:
Anonymous Author(s). 2017. Capabilities: Effects for Free. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Capabilities have been recently gaining new attention as a promising mechanism for controlling access to resources in systems and languages (Devriese et al. 2016; Dimoulas et al. 2014; Drossopoulou et al. 2007; Miller et al. 2003). A *capability* is an unforgeable token that can be used by its bearer to perform some operation on a resource (Dennis and Van Horn 1966). In a *capability-safe* language, all resources must be accessed through capabilities, and a resource-access capability must be obtained from a source that already has it: “only connectivity begets connectivity” (Miller et al. 2003). For example, a `logger` component that provides a logging service would need to be initialised with a capability providing the ability to append to the log file.

Capability-safe languages thus prohibit the *ambient authority* that is present in non-capability-safe languages. An implementation of a `logger` in OCaml or Java, for example, does not need to be passed a capability at initialisation time; it can simply import the appropriate file-access library and open the log file for appending itself. Critically, a malicious implementation of such a component could also delete the log, read from another file, or exfiltrate logging information over the network. Other mechanisms such as sandboxing can be used to limit the effects of such malicious components, but recent work has found that Java’s sandbox (for example) is difficult to use and is therefore often misused (Coker et al. 2015; Maass 2016).

In practice, reasoning about resource use in capability-based systems is mostly done informally. But if capabilities are useful for *informal* reasoning, shouldn’t they also aid in *formal* reasoning? Recent work by Drossopoulou et al. (2007) sheds some light on this question by presenting a logic that formalizes capability-based reasoning about trust between objects. Two other trains of work, rather than formalise capability-based reasoning itself, reason about how capabilities may be used. Dimoulas et al. (2014) developed a formalism for reasoning about which components may use a capability and which may influence (perhaps indirectly) the use of a capability. Devriese et al. (2016) formulate an effect parametricity theorem that limits the effects of an object based on the capabilities it possesses, and then use logical relations to reason about capability use in higher-order settings. Overall, this prior work presents new formal systems for reasoning about capability use, or reasoning about new properties using capabilities.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

We are interested in a different question: can capabilities be used to enhance formal reasoning that is currently done without relying on capabilities? In other words, what value do capabilities add to existing formal reasoning?

To answer this question, we decided to pick a simple and practical formal reasoning system, and see if capability-based reasoning could help. A natural choice for our investigation is effect systems (Nielson and Nelson 1999). Effect systems are a relatively simple formal reasoning approach, and keeping things simple will help to make the difference made by capabilities more obvious. Furthermore, effects have an intuitive link to capabilities: in a system that uses capabilities to protect resources, an expression can only have an effect on a resource if it is given a capability to do so.

How could capabilities help with effects? One challenge to the wider adoption of effect systems is their annotation overhead (Rytz et al. 2012). For example, Java's checked exception system is a kind of effect system, and is often criticised for being cumbersome (Kiniry 2006). Effect inference can be used to reduce the annotations required (Leijen 2014), but this has significant drawbacks: understanding error messages that arise through effect inference requires a detailed understanding of the internal structure of the code, not just its interface. Capabilities are a promising alternative for reducing the overhead of effect annotations, as suggested by the following example:

```
import log : String -> Unit with effect File.write

e
```

In the code above, written in a capability-safe language, what can we infer about the effects on resources (e.g. the file system or network) of evaluating `e`? Since we are in a capability-safe language, `e` has no ambient authority, and so the only way it can have any effect on resources is via the `log` function it imports. Note that this reasoning requires nothing about `e` other than that it obeys the rules of a capability-safe language; in particular, we don't require any effect annotations within `e`, and we don't need to analyse its structure as an effect inference would have to do. Also note that `e` might be arbitrarily large, perhaps consisting of an entire program that we have downloaded from a source that we trust enough to allow it to write to a log, but that we don't trust to access any other resources. Thus in this scenario, capabilities can be used to reason "for free" about the effect of a large body of code based on a few annotations on the components it imports.

The central intuition that we formalise in this paper is this: the effect of an unannotated expression can be given a bound based on the effects latent in variables that are in scope. Of course, there are challenges to solve on the way, most notably involving higher-order programs: how can we generalise this intuition if `log` takes a higher-order argument? If `e` evaluates not to unit but to a function, what can we infer about that function's effects?

In the remainder of this paper, we will formalise these ideas and explore these questions. To demonstrate, we introduce a pair of languages: the operation calculus `OC` (Section 3) and the capability calculus `CC` (Section 4). `OC` is a typed lambda calculus with a simple notion of capabilities and their operations, in which all code is effect-annotated. Relaxing this requirement, we then introduce `CC`, which permits the nesting of unannotated code inside annotated code in a controlled, capability-safe manner. A safe inference about the unannotated code can be made by inspecting the capabilities passed into it from its annotated surroundings. We then show how `CC` can model practical situations, presenting a range of examples to illustrate the benefits of a capability-flavoured effect system.

Throughout this paper we give motivating examples in a capability-safe language very similar to *Wyvern* as presented by Nistor et al. (2013). Wyvern features a first-class module system which distinguishes between pure and resourceful modules (Kurilova et al. 2016). We give several examples in Wyvern of interacting modules — some annotated, some unannotated — and demonstrate how they can be translated into our calculi to show how our type-and-effect system captures the properties of capability-based languages, and how it can aid in modular reasoning. A more thorough discussion of this translation is given in section 4. Several examples follow in section 5.

2 OPERATION CALCULUS (OC)

OC extends the simply-typed lambda calculus with a notion of primitive capabilities and their operations. Every function is annotated with the effects it may incur. Its static rules associate a type and a set of effects to well-formed programs. Defining OC will introduce the notations and concepts needed to understand CC, which allows developers to omit annotations from some expressions and uses capability-based reasoning to bound the effects of those expressions.

In a capability-safe language, "only connectivity begets connectivity" (Miller 2006): all access to a capability must derive from previous access. To prevent an infinite regress, there are a set of primitive capabilities passed into the program by the system environment. These primitive capabilities provide operations for manipulating resources in the system environment. For example, `File` might provide read/write operations on a particular file in the file system. For convenience, we often conflate primitive capabilities with the resources they manipulate, referring to both as resources. An effect in OC is a particular operation invoked on some resource; for example, `File.write`. Functions in an OC program are (conservatively) annotated with the effects they may incur when invoked. Annotations might be given in accordance with the principle of least authority to specify the maximum authority a component may exercise. When this authority is exceeded, an effect system like that of OC will reject the program, signaling an unsafe implementation. For example, consider the pair of modules[1] in Figure 1: the functor `logger` (declared with `module def`) must be instantiated with a `File` capability, and the resulting module exposes a single function `log`. The `client` module has a single function `run` which, when passed a `Logger`, will invoke `Logger.log`.

```
module def logger(f:{File}):Logger

def log(): Unit with {File.append} =
   f.read
```

```
module client

def run(l: Logger): Unit with {File.append} =
   l.log()
```

Fig. 1. The implementation of `logger.log` exceeds its specified authority.

`client.run` and `logger.log` are both annotated with {`File.append`}, but the (potentially malicious) implementation of `logger.log` incurs the `File.read` effect. In this section we develop rules for OC that can determine such mismatches between specification and implementation in annotated code.

OC makes some simplifying assumptions. The semantics of particular operations are not modeled — our only interest is in what operations could be invoked, and by whom. Therefore, we assume all operations are null-ary and return a dummy `unit` value; `File.write("hello, world!")` becomes `File.write`. Primitive capabilities and operations are fixed throughout run time and cannot be created or destroyed.

[1]Our formal grammar, below, does not include this *Wyvern*-like module syntax, but we can model the `logger` functor as a function and the `client` module as a record (which is itself encodable using functions). See section 4 for details.

2.1 Grammar (OC)

A grammar for OC programs is given in Figure 2. In addition to those from the lambda calculus, there are two new forms. A resource literal r is a variable drawn from a fixed set R. Resources model the primitive capabilities that the system passes into the program. `File` and `Socket` are examples of resource literals. An operation call $e.\pi$ is the invocation of an operation π on e. For example, invoking the open operation on the `File` resource would be `File`.open. Operations are drawn from a fixed set Π.

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

Fig. 2. Grammar for OC programs.

An effect is a pair $(r, \pi) \in R \times \Pi$. Sets of effects are denoted ε. As a shorthand, we write $r.\pi$ instead of (r, π). Effects should be distinguished from operation calls: an operation call is the invocation of a particular operation on a particular resource in a program, while an effect is a mathematical object describing this behaviour. The notation $r.*$ is a short-hand for the set $\{r.\pi \mid \pi \in \Pi\}$, which contains every effect on r. Sometimes we abuse notation by conflating the effect $r.\pi$ with the singleton $\{r.\pi\}$. We may also write things like $\{r_1.*, r_2.*\}$, which should be understood as the set of all operations on r_1 and r_2.

2.2 Semantics (OC)

During reduction an operation call may be evaluated. When this happens we say that a run time effect has taken place. Reflecting this, the form of the single-step reduction judgement is $e \longrightarrow e' \mid \varepsilon$, meaning e reduces to e', incurring the set of effects ε in the process. In the case of single-step reduction, ε is at most a single effect. Rules for single-step reductions are given in Figure 3.

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

$$
\frac{e_1 \longrightarrow e_1' \mid \varepsilon}{e_1 e_2 \longrightarrow e_1'\ e_2 \mid \varepsilon}\ (\text{E-App1})
\qquad
\frac{e_2 \longrightarrow e_2' \mid \varepsilon}{v_1\ e_2 \longrightarrow v_1\ e_2' \mid \varepsilon}\ (\text{E-App2})
\qquad
\frac{}{(\lambda x : \tau.e)v_2 \longrightarrow [v_2/x]e \mid \varnothing}\ (\text{E-App3})
$$

$$
\frac{e \rightarrow e' \mid \varepsilon}{e.\pi \longrightarrow e'.\pi \mid \varepsilon}\ (\text{E-OperCall1})
\qquad
\frac{}{r.\pi \longrightarrow \mathtt{unit} \mid \{r.\pi\}}\ (\text{E-OperCall2})
$$

Fig. 3. Single-step reductions in OC.

The first three rules are analogous to reductions in the lambda calculus. E-App1 and E-App2 incur the effects of reducing their subexpressions. E-App3 replaces free occurrences of the formal name x in e with the actual value v_2 being passed as an argument, which incurs no effects. The notation for this is $[v_2/x]e$. It is significant that variables are only substituted for values: if x is replaced by an arbitrary expression, the substitution could be introducing arbitrary effects. However, values incur no effects, so replacing x by a value will not introduce any extra effects. Thus OC is a call-by-value language.

The first new rule is E-OperCall1, which reduces the receiver of an operation call; the effects incurred are the effects incurred by reducing the receiver. When an operation π is invoked on a resource literal r, E-OperCall2 will reduce it to `unit`, incurring $\{r.\pi\}$ as a result. For example, `File.write` $\longrightarrow$ `unit` $\mid$ $\{$`File.write`$\}$ by E-OperCall2. `unit` can be treated as a derived form; an explanation is given in section 4.

A multi-step reduction is a sequence of zero or more single-step reductions. The resulting set of run time effects is the union of all the run time effects from the intermediate single-steps. Rules for multi-step reductions are given in Figure 4. By E-MultiStep1, any expression can "reduce" to itself with no run time effects. By E-MultiStep2, any single-step reduction is also a multi-step reduction. If $e \longrightarrow e' \mid \varepsilon_1$ and $e' \longrightarrow e'' \mid \varepsilon_2$ are sequences of reductions, then so is $e \longrightarrow e'' \mid \varepsilon_1 \cup \varepsilon_2$, by E-MultiStep3.

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

$$\frac{}{e \longrightarrow^* e \mid \varnothing}\ (\text{E-MultiStep1}) \qquad \frac{e \longrightarrow e' \mid \varepsilon}{e \longrightarrow^* e' \mid \varepsilon}\ (\text{E-MultiStep2}) \qquad \frac{e \longrightarrow^* e' \mid \varepsilon_1 \quad e' \longrightarrow^* e'' \mid \varepsilon_2}{e \longrightarrow^* e'' \mid \varepsilon_1 \cup \varepsilon_2}\ (\text{E-MultiStep3})$$

Fig. 4. Multi-step reductions in OC.

2.3 Static Rules (OC)

A grammar for types, contexts, and sets of effects is given in Figure 5. The base types of OC are sets of resources, denoted $\{\bar{r}\}$. If an expression e is associated with type $\{\bar{r}\}$, it means e will reduce to one of the literals in $\bar{r}$ (assuming e terminates). The set of empty resources (denoted $\varnothing$) is also a valid type, but has no inhabitants. There is a single type constructor $\rightarrow_\varepsilon$, where ε is a concrete set of effects. $\tau_1 \rightarrow_\varepsilon \tau_2$ is the type of a function which takes a τ_1 as input, returns a τ_2 as output, and whose body incurs no more than those effects in ε. ε is a conservative bound: if an effect $r.\pi \in \varepsilon$, it is not guaranteed to happen at run time, but if $r.\pi \notin \varepsilon$, it cannot happen at run time. A typing context Γ maps variables to types.

$$\begin{array}{lcll} \tau & ::= & & \textit{types}: \\ & \mid & \{\bar{r}\} & \textit{resource set} \\ & \mid & \tau \rightarrow_\varepsilon \tau & \textit{annotated arrow} \\ \\ \varepsilon & ::= & & \textit{effects}: \\ & \mid & \{\overline{r.\pi}\} & \textit{effect set} \end{array} \qquad \begin{array}{lcll} \Gamma & ::= & & \textit{type ctx}: \\ & \mid & \varnothing & \textit{empty ctx.} \\ & \mid & \Gamma, x : \tau & \textit{var. binding} \end{array}$$

Fig. 5. Grammar for types in OC.

To illustrate the types of some functions, if $\texttt{log}_1$ has the type $\{$`File`$\}$ $\rightarrow_{\{\texttt{File.append}\}}$ `Unit`, then invoking $\texttt{log}_1$ will either incur `File.append` or no effects. If $\texttt{log}_2$ has the type $\{$`File`$\}$ $\rightarrow_{\{\texttt{File.*}\}}$ `Unit`, then invoking $\texttt{log}_2$ could incur any effect on `File`, or no effects.

Knowing approximately what effects a piece of code may incur helps a developer determine whether it can be trusted. For example, consider $\texttt{log}_3 = \lambda f : \{\texttt{File}\}.\ e$, which is a logging function that takes a `File` as an argument and then executes e. Suppose this function were to typecheck as $\{$`File`$\}$ $\rightarrow_{\{\texttt{File.*}\}}$ `Unit` — seeing that invoking this function could incur any effect on `File`, and not just its expected least authority `File.append`, a developer may therefore decide this implementation cannot be trusted and choose not to execute it. In this spirit, the static rules of OC associate well-typed programs with a type and a set of effects: the judgement $\Gamma \vdash e : \tau$ `with` ε, means e will reduce to a term of type τ (assuming it terminates), incurring no more effects than those in ε. Judgements are given in Figure 6.

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

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

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

$$\frac{\Gamma \vdash e : \{\bar{r}\} \texttt{ with } \varepsilon}{\Gamma \vdash e.\pi : \texttt{Unit with } \varepsilon \cup \{\bar{r}.\pi\}}\ (\varepsilon\text{-OperCall}) \qquad \frac{\Gamma \vdash e : \tau \texttt{ with } \varepsilon \quad \tau <: \tau' \quad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash e : \tau' \texttt{ with } \varepsilon'}\ (\varepsilon\text{-Subsume})$$

Fig. 6. Type-with-effect judgements in OC.

ε-Var approximates the run time effects of a variable as $\varnothing$. ε-Resource does the same for resource literals. Though a resource captures several effects (namely, every possible operation on itself), attempting to "reduce" a resource will incur no effects; something must be done with the resource, such as an operation call, in order to have an effect. For a similar reason, ε-Abs approximates the effects of a function literal as $\varnothing$, and ascribes an arrow type annotated with those effects captured by the function. ε-App approximates a lambda application as incurring those effects from evaluating the subexpressions and the effects incurred by executing the body of the function to which the left-hand side evaluates. The effects of the function body are taken from the function's arrow type. An operation call on a resource literal reduces to `unit`, so ε-OperCall ascribes its type as `Unit`. The approximate effects of an operation call are: the effects of reducing the subexpression, and then the operation π on every possible resource which that subexpression to which that subexpression might reduce. For example, consider $e.\pi$, where $\Gamma \vdash e : \{\texttt{File}, \texttt{Socket}\} \texttt{ with } \varnothing$. Then e could evaluate to `File`, in which case the actual run time effect is $\texttt{File}.\pi$, or it could evaluate to `Socket`, in which case the actual run time effect is $\texttt{Socket}.\pi$. Determining which will happen is, in general, undecidable; the safe approximation is to treat them both as happening. The last rule ε-Subsume produces a new judgement by widening the type or approximate effects on an existing one. Subtyping judgements are given in Figure 7.

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

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

Fig. 7. Subtyping judgements of OC.

S-Arrow is the standard rule for arrow types, but also stipulates that the effects on the arrow of the subtype must be contained in the effects on the arrow of the supertype: a valid subtype should not invoke any effects the supertype does not already know about. S-Resource says that a subset of resources is a subtype. To illustrate, consider $\{\bar{r}_1\} <: \{\bar{r}_2\}$ — any value with type $\{\bar{r}_1\}$ can reduce to any resource literal in $\bar{r}_1$, so to be compatible with an interface $\{\bar{r}_2\}$, the resource literals in $\bar{r}_1$ must also be in $\bar{r}_2$.

These rules let us determine what sort of effects might be incurred when a piece of code is executed. For example, consider $rw = \lambda x : \{\texttt{File}, \texttt{Socket}\}.\ \texttt{x.write}$, which takes either a `File` or a `Socket` and writes to it. If

rw is applied, it could incur `Socket.write` or `File.write`, depending on what had been passed. In general, there is no way to statically determine what this will be, so the safe approximation is $\{\texttt{File.write}, \texttt{Socket.write}\}$. This is the approximation given by a judgement like $\vdash rw\ \texttt{File} : \texttt{Unit with}\ \{\texttt{File.write}, \texttt{Socket.write}\}$. A derivation of this judgement is given in Figure 8. To fit on the page, all resources and operations have been abbreviated to their first letter. A developer who only expects rw to be incurring `File.write` can typecheck rw, see that it could also be writing to `Socket`, and decide it should not be used. If client code was annotated with $\{\texttt{File.write}\}$ and tried to use this function, the type system would reject it.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \{\mathsf{F}, \mathsf{S}\} \vdash x : \{\mathsf{F}, \mathsf{S}\}}\ (\varepsilon\text{-Var})}{x : \{\mathsf{F}, \mathsf{S}\} \vdash \mathsf{x.w} : \mathsf{Unit\ with}\ \{\mathsf{F.w}, \mathsf{S.w}\}}\ (\varepsilon\text{-OperCall})}{\lambda x : \{\mathsf{F}, \mathsf{S}\}.\ \mathsf{x.w} : \{\mathsf{F}, \mathsf{S}\} \rightarrow_{\{\mathsf{F.w}, \mathsf{S.w}\}} \mathsf{Unit\ with}\ \varnothing}\ (\varepsilon\text{-Abs}) \qquad \dfrac{\dfrac{}{\vdash \mathsf{F} : \{\mathsf{F}\}\ \mathsf{with}\ \varnothing}\ (\varepsilon\text{-Resource}) \qquad \dfrac{F \in \{F, S\}}{\{F\} <: \{F, S\}}\ (\text{S-Resource})}{\vdash \mathsf{F} : \{\mathsf{F}, \mathsf{S}\}}\ (\varepsilon\text{-Subsume})}{\vdash (\lambda x : \{\mathsf{F}, \mathsf{S}\}.\ \mathsf{x.w})\ \mathsf{F} : \mathsf{Unit\ with}\ \{\mathsf{F.w}, \mathsf{S.w}\}}\ (\varepsilon\text{-App})$$

Fig. 8. Derivation tree for $\vdash rw\ \texttt{File} : \texttt{Unit with}\ \{\texttt{File.write}, \texttt{Socket.write}\}$.

2.4 Soundness (OC)

To show the rules of OC are sound requires an appropriate notion of static approximations being safe with respect to the reductions. If a judgement like $\Gamma \vdash e : \tau\ \texttt{with}\ \varepsilon$ were correct, successive reductions on e should never incur effects not in ε. Furthermore, as e is reduced, we learn more about what it is, so approximations on the reduced forms can only get more specific; compare this with how the type of reduced expressions can only get more specific. Adding this to the standard definition of soundness yields the following theorem statement.

Theorem 2.1 (OC Single-step Soundness). *If $\Gamma \vdash e_A : \tau_A\ \texttt{with}\ \varepsilon_A$ and e_A is not a value or variable, then $e_A \longrightarrow e_B \mid \varepsilon$, where $\Gamma \vdash e_B : \tau_B\ \texttt{with}\ \varepsilon_B$ and $\tau_B <: \tau_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, for some $e_B, \varepsilon, \tau_B, \varepsilon_B$.*

Our approach to proving soundness is to show progress and preservation. Noting that the rules for values give $\varnothing$ as their approximate effects, the proof of the progress theorem is routine.

Theorem 2.2 (OC Progress). *If $\Gamma \vdash e : \tau\ \texttt{with}\ \varepsilon$ and e is not a value or variable, then $e \longrightarrow e' \mid \varepsilon'$, for some $e', \varepsilon' \subseteq \varepsilon$.*

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

To show preservation we need to know that effect safety is preserved by the substitution in E-App3. The semantics are call-by-value, so the name of a function argument is only ever replaced with a value, and we know that the approximate effects of values are $\varnothing$, so the substitution does not introduce more effects. Beyond this observation, the proof is routine.

Theorem 2.3 (OC Preservation). *If $\Gamma \vdash e_A : \tau_A\ \texttt{with}\ \varepsilon_A$ and $e_A \longrightarrow e_B \mid \varepsilon$, then $\Gamma \vdash e_B : \tau_B\ \texttt{with}\ \varepsilon_B$, where $\tau_B <: \tau_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, for some $e_B, \varepsilon, \tau_B, \varepsilon_B$.*

Proof. By induction on the derivations of $\Gamma \vdash e_A : \tau_A\ \texttt{with}\ \varepsilon_A$ and $e_A \longrightarrow e_B \mid \varepsilon$. □

The single-step soundness theorem now holds by combining progress and preservation. The soundness of multi-step reductions follows by inducting on the length of a multi-step and appealing to single-step soundness.

Theorem 2.4 (OC Single-step Soundness). *If $\Gamma \vdash e_A : \tau_A\ \texttt{with}\ \varepsilon_A$ and e_A is not a value or variable, then $e_A \longrightarrow e_B \mid \varepsilon$, where $\Gamma \vdash e_B : \tau_B\ \texttt{with}\ \varepsilon_B$ and $\tau_B <: \tau_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, for some $e_B, \varepsilon, \tau_B, \varepsilon_B$.*

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

Theorem 2.5 (OC Multi-step Soundness). *If* $\Gamma \vdash e_A : \tau_A$ `with` ε_A *and* $e_A \longrightarrow^* e_B \mid \varepsilon$*, then* $\Gamma \vdash e_B : \tau_B$ `with` ε_B*, where* $\tau_B <: \tau_A$ *and* $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$.

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

3 CAPABILITY CALCULUS (CC)

OC requires every function to be annotated. The verbosity of such effect systems has been given as a reason for why they have not seen widespread use (Rytz et al. 2012) — if we relax the requirement that all code be annotated, can a type system say anything useful about the parts which are not? Allowing a mix of annotated and unannotated code helps reduce the cognitive overhead on developers, allowing them to rapidly prototype in the unannotated sublanguage and incrementally add annotations as they are needed. However, reasoning about unannotated code is difficult in general. Figure 9 demonstrates why: someMethod takes a function f as input and executes it, but the effects of f depend on its implementation. Without more information, there is no way to know what effects might be incurred by someMethod.

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

Fig. 9. What effects can someMethod incur?

A capability-safe design can help us: because the only authority code can exercise is that which is explicitly given to it, the only capabilities that the unannotated code can use must be passed into it. If these capabilities are being passed in from an annotated environment, we know what effects they capture. These effects are therefore a conservative upper bound on what can happen in the unannotated code. To demonstrate, consider a developer who wants to decide whether to use the logger functor in Figure 10. It must be instantiated with two capabilities, File and Socket, and provides an unannotated function log.

```
module def logger(f:{File},s:{Socket}):Logger

def log(x: Unit): Unit
  ...
```

Fig. 10. In a capability-safe setting, logger can only exercise authority over the File and Socket capabilities given to it.

What effects will be incurred if Logger.log is invoked? One approach is to manually[2] examine its source code, but this is tedious and error-prone. In many real-world situations, the source code may be obfuscated or unavailable. A capability-based argument can do better: the only authority which Logger can exercise is that which it has been explicitly given. Here, the Logger requires a File and a Socket, so {File.∗, Socket.∗} is an upper bound on the effects of Logger. Knowing Logger could be performing arbitrary reads and writes to File, or arbitrary communication with the Socket, the developer decides this implementation cannot be trusted and does not use it.

The reasoning we employed only required us to examine the interface of the unannotated code for the capabilities passed into it. To model this situation in CC, we add a new import expression that selects what

[2]or automatically—but if the automation produces an unexpected result we must fall back to manual reasoning to understand why.

authority ε the unannotated code may exercise. In the above example, the expected least authority of `Logger` is {`File.append`}, so that is what the corresponding `import` would select. The type system can then check if the capabilities being passed into the unannotated code exceed its selected authority. If it accepts, then ε safely approximates the effects of the unannotated code. This is the key result: when unannotated code is nested inside annotated code, capability-safety enables us to make a safe inference about its effects by examining what capabilities are being passed in by the annotated code.

3.1 Grammar (CC)

The grammar of CC is split into rules for annotated code and analogous rules for unannotated code. To distinguish the two, we put a hat above annotated types, expressions, and contexts: $\hat{e}$, $\hat{\tau}$, and $\hat{\Gamma}$ are annotated, while e, τ, and Γ are unannotated. The rules for unannotated programs and their types are given in Figure 11. They are much the same as in OC, but the type constructor $\rightarrow$ is not annotated with a set of effects: the type $\tau_1 \rightarrow \tau_2$ says nothing about what effects may or may not happen when the function is executed. Unannotated types τ are built using $\rightarrow$ and sets of resources $\{\bar{r}\}$. An unannotated context Γ maps variables to unannotated types.

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

Fig. 11. Unannotated programs and types in CC.

Rules for annotated programs and their types are given in Figure 12. Except for the new `import` expression, the rules are identical to those in OC, except now everything has a hat above it.

$$
\begin{array}{llll}
\hat{e} & ::= & & \mathit{labeled\ exprs:} \\
 & | & x & \\
 & | & \hat{v} & \\
 & | & \hat{e}\ \hat{e} & \\
 & | & \hat{e}.\pi & \\
 & | & \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e & \mathit{import} \\
 & & & \\
\hat{v} & ::= & & \mathit{labeled\ values:} \\
 & | & r & \\
 & | & \lambda x : \hat{\tau}.\hat{e} &
\end{array}
\qquad
\begin{array}{llll}
\hat{\tau} & ::= & & \mathit{annotated\ types:} \\
 & | & \{\bar{r}\} & \\
 & | & \hat{\tau} \rightarrow_{\varepsilon} \hat{\tau} & \\
 & & & \\
\hat{\Gamma} & ::= & & \mathit{annotated\ type\ ctx:} \\
 & | & \varnothing & \\
 & | & \hat{\Gamma}, x : \hat{\tau} & \\
 & & & \\
\varepsilon & ::= & & \mathit{effects:} \\
 & | & \{\overline{r.\pi}\} & \mathit{effect\ set}
\end{array}
$$

Fig. 12. Annotated programs and types in CC.

The new form is $\texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e$, modelling the points at which capabilities are passed from annotated code into unannotated code. e is the unannotated code. $\hat{e}$ is the capability being given to it; we call $\hat{e}$ an import. For simplicity, we assume only one capability is being passed into e. $\hat{e}$ is associated with the name x inside e. ε_s is the maximum authority that e is allowed to exercise (its "selected authority"). As an example, suppose an unannotated `Logger`, which requires `File`, is expected to only append to a file, but has an implementation that writes. This would be modelled by the expression $\texttt{import}(\texttt{File.append})\ x = \texttt{File}\ \texttt{in}\ \lambda y : \texttt{Unit}.\ \texttt{x.write}$.

`import` is the only way to mix annotated and unannotated code, because it is the only situation in which we can say something interesting about the unannotated code. For the rest of our discussion on CC, we will only be interested in unannotated code when it is encapsulated by an `import` expression.

One of the requirements of capability safety is there be no ambient authority. This requirement is met by forbidding resource literals r from being used directly inside an `import` statement (they can still be passed in as a capability via the `import`'s binding variable x). We could enforce this syntactically, by removing r from the language of unannotated expressions, but we choose to do it instead using the typing rule for `import`, given below.

3.2 Semantics (CC)

Reductions are defined on annotated expressions. Excluding `import`, the annotated sublanguage of CC is the same as OC, so we take every reduction rule of OC as a valid reduction rule in CC. For brevity, they are not restated.

If unannotated code e is wrapped inside annotated code $\texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e$, we transform it into annotated code by recursively annotating its parts with ε_s. In practice, it is meaningful to execute purely unannotated code — but our only interest is when that code is wrapped inside an `import` expression, so we do not bother to give rules for it. There are two new rules for reducing `import` expressions, given in Figure 13: E-Import1 reduces the capability being imported, while E-Import2 first annotates e with its selected authority ε — this is $\texttt{annot}(e, \varepsilon)$ — and then substitutes the import $\hat{v}$ for its name x in e — this is $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$.

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

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

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

Fig. 13. New single-step reductions in CC.

$\texttt{annot}(e, \varepsilon)$ produces the expression obtained by recursively annotating the parts of e with the set of effects ε. A definition is given in Figure 14. There are versions of `annot` defined for expressions and types. Later we shall need to annotate contexts, so the definition is given here. It is worth mentioning that `annot` operates on a purely syntatic level — nothing prevents us from annotating a program with something unsafe, so any use of `annot` must be justified.

3.3 Static Rules (CC)

A term can be annotated or unannotated, so we need to be able to recognise when either is well-typed. We do not reason about the effects of unannotated code directly, so judgements about them have the form $\Gamma \vdash e : \tau$. Subtyping judgements have the form $\tau <: \tau$. A summary of the rules for unannotated judgements is given in Figure 15. Each is analogous to some rule in OC, but the parts relating to effects have been removed.

$$\mathsf{annot} :: e \times \varepsilon \to \hat{e}$$

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

$$\mathsf{annot} :: \tau \times \varepsilon \to \hat{\tau}$$

$$\begin{aligned}
&\mathsf{annot}(\{\bar{r}\}, \_) = \{\bar{r}\} \\
&\mathsf{annot}(\tau_1 \to \tau_2, \varepsilon) = \mathsf{annot}(\tau_1, \varepsilon) \to_{\varepsilon} \mathsf{annot}(\tau_2, \varepsilon).
\end{aligned}$$

$$\mathsf{annot} :: \Gamma \times \varepsilon \to \hat{\Gamma}$$

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

Fig. 14. Definition of annot.

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

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

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_3}\ (\text{T-App}) \qquad \frac{\Gamma \vdash e : \{\bar{r}\}}{\Gamma \vdash e.\pi : \mathtt{Unit}}\ (\text{T-OperCall})$$

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

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

Fig. 15. (Sub)typing judgements for the unannotated sublanguage of CC

Since the annotated subset of CC contains OC, all the OC judgements apply, but now we put hats on everything to signify that a typing judgement is being made about annotated code inside an annotated context. This looks like $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ with ε. Except for notation the judgements are the same, so we shall not repeat them. The only new rule is ε-Import, given in Figure 24, which gives the type and approximate effects of an `import` expression. This is the only way to reason about what effects might be incurred by some unannotated code. The rule is complicated, so to explain it we shall start with a simplified version and spend the rest of this section building up to the final version of ε-Import.

To begin, typing $\mathtt{import}(\varepsilon_s)\ x = \hat{e}\ \mathtt{in}\ e$ in a context $\hat{\Gamma}$ requires us to know that the import $\hat{e}$ is well-typed, so we add the premise $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}$ with ε_1. Since $x = \hat{e}$ is an import, it can be used throughout e. We do not want e to exercise authority it hasn't explicitly selected, so whatever capabilities it uses must be selected by the `import` expression; therefore, we require that e can be typechecked using only the binding $x : \hat{\tau}$. There is a problem though: e is unannotated and $\hat{\tau}$ is annotated, and there is no rule for typechecking unannotated code in an annotated context. To get around this, we define a function `erase` in Figure 16 which removes the annotations from a type. We then add $x : \mathtt{erase}(\hat{\tau}) \vdash e : \tau$ as a premise.

Note that, since the environment Γ for e has only one binding (for x), it cannot contain any bindings of resource literals—and the rule T-Resource requires a binding in the environment in order to type a resource literal in an expression. Typing e in the restricted environment given by `import` thus prohibits ambient authority.

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

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

Fig. 16. Definition of erase.

The first version of ε-Import is given in Figure 17. Since $\texttt{import}(\varepsilon_s)\ x = \hat{v}\ \texttt{in}\ e \longrightarrow [\hat{v}/x]\texttt{annot}(e, \varepsilon_s)$ by E-Import2, the ascribed type is $\texttt{annot}(\tau, \varepsilon)$, which is the type of the unannotated code, annotated with its selected authority ε_s. The effects of the `import` are $\varepsilon_1 \cup \varepsilon_s$ — the former comes from reducing the imported capability, which happens before the body of the `import` is annotated and executed, and the latter contains all the effects which the unannotated code might incur.

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s \cup \varepsilon_1}\ (\varepsilon\text{-Import1-Bad})$$

Fig. 17. A first (incorrect) rule for type-and-effect checking `import` expressions.

At the moment there is no relation between the selected authority ε and those effects captured by the imported capability $\hat{e}$. Consider $\hat{e}' = \texttt{import}(\varnothing)\ x = \texttt{File}\ \texttt{in}\ \texttt{x.write}$, which imports a `File` and writes to it, but declares its authority as $\varnothing$. According to ε-Import1, $\vdash \hat{e}' : \texttt{Unit}\ \texttt{with}\ \varnothing$, but this is clearly wrong since $\hat{e}'$ writes to `File`. An `import` should only be well-typed if the capability being imported only captures effects contained in the unannotated code's selected authority ε. In this case, `File` captures $\{\texttt{File}.*\}$, which is not contained in the selected authority $\varnothing$, so it should be rejected for that reason. To this end we define a function `effects`, which collects the set of effects that an annotated type captures. A first (but not yet correct) definition is given in Figure 18. We can then add the premise $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s$ to require that any imported capability must not capture authority beyond that selected in ε_s. The updated rule is given in Figure 19.

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

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

Fig. 18. A first (incorrect) definition of `effects`.

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ (\varepsilon\text{-Import2-Bad})$$

Fig. 19. A second (still incorrect) rule for type-and-effect checking `import` expressions.

The counterexample from before is now rejected by ε-Import2, but there are still issues: the annotations on one import can be broken by another import. To illustrate, consider Figure 20 where two[3] capabilities are imported. This program imports a function go which, when given a $\texttt{Unit} \to_{\varnothing} \texttt{Unit}$ function with no effects, will execute it. The other import is `File`. The unannotated code creates a $\texttt{Unit} \to \texttt{Unit}$ function which writes to `File` and passes it to go, which subsequently incurs `File.write`.

[3]Our formalisation only permits a single capability to be imported, but this discussion leads to a generalisation needed for the rules to be safe when multiple capabilities can be imported. In any case, importing multiple capabilities can be handled with an encoding of pairs.

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

Fig. 20. Permitting multiple imports will break ε-Import2.

In the world of annotated code it is not possible to pass a file-writing function to go, but because the judgement $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$ discards the annotations on go, and since the file-writing function has type `unit` → `unit`, the unannotated world accepts it. The approximation is actually safe at the top-level, because the `import` selects `{File.∗}`, which contains `File.write` — but it contains code that violates the type signature of go. We want to prevent this.

If go had the type $\texttt{Unit} \rightarrow_{\{\texttt{File.write}\}} \texttt{Unit}$ the above example would be safe, but a modified version where a file-reading function is passed to go would have the same issue. go is only safe when it expects every effect that the unannotated code might pass to it: if go had the type $\texttt{Unit} \rightarrow_{\{\texttt{File.*}\}} \texttt{Unit}$, then the unannotated code cannot pass it a capability with an effect it isn't already expecting, so the annotation on go cannot be violated. Therefore, we require imported capabilities to have authority to incur the effects in ε. To achieve greater control in how we say this, the definition of `effects` is split into two separate functions called `effects` and `ho-effects`. The latter is for higher-order effects, i.e. the effects that are not captured within a function, but rather are possible because of what it is passed as an argument. If values of $\hat{\tau}$ possess a capability that can be used to incur the effect $r.\pi$, then $r.\pi \in \texttt{effects}(\hat{\tau})$. If values of $\hat{\tau}$ can incur an effect $r.\pi$, but need to be given the capability (as a function argument) by someone else in order to do it, then $r.\pi \in \texttt{ho-effects}(\hat{\tau})$. Definitions are given in Figure 21.

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

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

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

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

Fig. 21. Effect functions (corrected).

`effects` and `ho-effects` are mutually recursive, with base cases for resource types. Any effect can be directly incurred by a resource on itself, hence $\texttt{effects}(\{\bar{r}\}) = \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$. A resource cannot be used to indirectly invoke some other effect, so $\texttt{ho-effects}(\{\bar{r}\}) = \varnothing$. The mutual recursion echoes the subtyping rule for functions. Recall that functions are contravariant in their input type and covariant in their output; likewise, both functions recurse on the input-type using the other function, and recurse on the output-type using the same function.

In light of these new definitions, we still require $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s$ — unannotated code must select any effect its capabilities can incur — but we add a new premise $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau})$, stipulating that imported capabilities must know about every effect they could be given by the unannotated code (which is at most ε). The counterexample from Figure 20 is now rejected, because $\texttt{ho-effects}((\texttt{Unit} \rightarrow_{\varnothing} \texttt{Unit}) \rightarrow_{\varnothing} \texttt{Unit}) = \varnothing$, but $\{\texttt{File.*}\} \not\subseteq \varnothing$. However, this is *still* not sufficient! Consider $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau}_1 \rightarrow_{\varepsilon'} \hat{\tau}_2)$. We want *every* higher-order capability involved to be expecting ε_s. Expanding the definition of `ho-effects`, this is the same as $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$. Let $r.\pi \in \varepsilon_s$ and suppose $r.\pi \in \texttt{effects}(\hat{\tau}_1)$, but $r.\pi \notin \texttt{ho-effects}(\hat{\tau}_2)$. Then $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$ is still true, but $\hat{\tau}_2$ is not expecting $r.\pi$. Unannotated code could then violate the annotations on $\hat{\tau}_2$ by passing it a capability for $r.\pi$, using the same trickery as before. The cause of the issue is that $\subseteq$ does not distribute over $\cup$. We want a relation like $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$, which also implies $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau}_1)$ and $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau}_2)$. Figure 22 defines this: safe is a distributive version of $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau})$ and ho-safe is a distributive version of $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau})$.

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

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

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

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

Fig. 22. Safety judgements in CC.

An amended version of ε-Import is given in Figure 23. It contains a new premise $\texttt{ho-safe}(\hat{\tau}, \varepsilon_s)$ which formalises the notion that every capability which could be given to a value of $\hat{\tau}$ — or any of its constituent pieces — must be expecting the effects ε_s it might be given by the unannotated code.

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \ \texttt{with}\ \varepsilon_1 \ \ \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s \\ \texttt{ho-safe}(\hat{\tau}, \varepsilon_s) \ \ x : \texttt{erase}(\hat{\tau}) \vdash e : \tau\end{array}}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ (\varepsilon\textsc{-Import3-Bad})$$

Fig. 23. A third (still incorrect) rule for type-and-effect checking import expressions.

The premises so far restrict what authority can be selected by unannotated code, but what about authority passed as a function argument? Consider the example $\hat{e} = \texttt{import}(\varnothing)\ x = \texttt{unit in}\ \lambda f : \texttt{File. f.write}$. The unannotated code selects no capabilities and returns a function which, when given File, incurs File.write. This satisfies the premises in ε-Import3, but its annotated type is $\{\texttt{File}\} \to_{\varnothing} \texttt{Unit}$ — not good!

Suppose the unannotated code defines a function f, which gets annotated with ε_s to produce $\texttt{annot}(f, \varepsilon_s)$. Suppose $\texttt{annot}(f, \varepsilon_s)$ is invoked at a later point in the annotated world and incurs the effect $r.\pi$. What is the source of $r.\pi$? If $r.\pi$ was selected by the import expression surrounding f, it is safe for $\texttt{annot}(f, \varepsilon_s)$ to incur this effect. Otherwise, $\texttt{annot}(f, \varepsilon_s)$ may have been passed an argument which can be used to incur $r.\pi$, in which case $r.\pi$ is a higher-order effect of $\texttt{annot}(f, \varepsilon_s)$. If the argument is a function, then $r.\pi \in \varepsilon_s$ by the soundness of OC (or it would not typecheck). If the argument is a resource r, then $\texttt{annot}(f, \varepsilon_s)$ may exercise $r.\pi$ without declaring it — this is the case we do not yet account for.

We want ε_s to contain every effect captured by resources passed into $\texttt{annot}(f, \varepsilon_s)$ as arguments. We can do this by inspecting the (unannotated) type of f for resource sets. For example, if the unannotated type is $\{\texttt{File}\} \to \texttt{Unit}$, then we need $\{\texttt{File}.*\}$ in ε_s. To do this, we add a new premise $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s$. ho-effects is only defined on annotated types, so we first annotate τ with $\varnothing$. We are only inspecting the resources passed into f as arguments, so the annotations are not relevant – annotating τ with $\varnothing$ is therefore a good choice. We can now handle the example from before. The unannotated code types via the judgement $x : \texttt{Unit} \vdash \lambda f : \{\texttt{File}\}.\ \texttt{f.write} : \{\texttt{File}\} \to \texttt{Unit}$. Its higher-order effects are $\texttt{ho-effects}(\texttt{annot}(\{\texttt{File}\} \to \texttt{Unit}, \varnothing)) = \{\texttt{File}.*\}$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$, so the example is safely rejected.

The final version of ε-Import is given in Figure 24. With it, we can now model the example from the beginning of this section, where the `Logger` selects the `File` capability and exposes an unannotated function `log` with type $\texttt{Unit} \to \texttt{Unit}$ and implementation e. The expected least authority of `Logger` is $\{\texttt{File.append}\}$, so its corresponding `import` expression would be $\texttt{import}(\texttt{File.append})\ f = \texttt{File in } \lambda x : \texttt{Unit}.\ e$. The imported capability is $f = \texttt{File}$, and $\texttt{effects}(\{\texttt{File}\}) = \{\texttt{File}.*\} \not\subseteq \{\texttt{File.append}\}$, so this example is safely rejected: `Logger.log` has authority to do anything with `File`, and its implementation e might be violating its stipulated least authority $\{\texttt{File.append}\}$.

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

Fig. 24. The final rule for typing imports.

3.4 Soundness (CC)

Only annotated programs can be reduced and have their effects approximated, so the soundness theorem only applies to annotated judgements. Its statement is given below.

Theorem 3.1 (CC Single-step Soundness). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A \texttt{ with } \varepsilon_A$ and $\hat{e}_A$ is not a value, then $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$, where $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B \texttt{ with } \varepsilon_B$ and $\hat{\tau}_B <: \hat{\tau}_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, for some $\hat{e}_B, \varepsilon, \hat{\tau}_B, \varepsilon_B$.*

Because the rules of `OC`, proven sound in section 2, are also rules of `CC`, we do not repeat them here. The progress theorem has a new case for when the typing rule used is ε-Import, but the proof is routine.

Theorem 3.2 (CC Progress). *If $\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon$ and $\hat{e}$ is not a value, then $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon'$, for some $\hat{e}', \varepsilon' \subseteq \varepsilon$.*

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

The preservation theorem also has an extra case for when the typing rule used is ε-Import, with two subcases, depending on whether the reduction rule used was E-Import1 and E-Import2. The former is straightforward, but the latter is tricky; we need several lemmas to do it. Firstly, since ε_s is an upper bound on what effects can be incurred by the unannotated code, it should also be an upper bound on what effects can be incurred by the capabilities passed into the unannotated code. Therefore, if we take $\hat{\tau}$ and replace its annotations with ε_s, we should get a more general function type $\texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon_s)$. This result is given as the pair of lemmas below.

Lemma 3.3 (CC Approximation 1). *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 3.4 (CC Approximation 2). *If $\texttt{ho-effects}(\hat{\tau}) \subseteq \varepsilon$ and $\texttt{safe}(\hat{\tau}, \varepsilon)$ then $\texttt{annot}(\texttt{erase}(\hat{\tau}), \varepsilon) <: \hat{\tau}$.*

Proof. By simultaneous induction on derivations of $\texttt{ho-safe}(\hat{\tau}, \varepsilon)$ and $\texttt{safe}(\hat{\tau}, \varepsilon)$. □

Recall that function types are contravariant in their input, so the subtyping and subsetting relations flip direction when considering the input type of a function. This is why there are two lemmas: one for each direction.

Now, if E-Import2 is applied, the reduction has the form $\texttt{import}(\varepsilon_s)\ x = \hat{v}_i \texttt{ in } e \longrightarrow [\hat{v}_i/x]\texttt{annot}(e, \varepsilon_s) \mid \varnothing$. Since $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$, it is reasonable to expect (1) $\hat{\Gamma} \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s$ is true — the reduction annotates e with ε_s, so the type after annotation ought to be the type of e, annotated with ε_s, i.e. $\texttt{annot}(\tau, \varepsilon_s)$. Furthermore, $\texttt{annot}(e, \varepsilon_s)$ has the same structure as e — the annotations do not change what capabilities can be used, so the bound ε_s on the authority of e also bounds the authority of $\texttt{annot}(e, \varepsilon_s)$. Now, if judgement (1) holds, then $\hat{\Gamma} \vdash [\hat{v}_i/x]\texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon_s$ would hold by the substitution lemma (remembering we only substitute values, as not to introduce extra effects). That judgement (1) does hold is the subject of the following lemma.

Lemma 3.5 (CC Annotation). *If the following are true:*

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

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

The premises of the lemma are very specific to the premises of ε-Import, but generalised to accommodate a proof by induction: e is allowed to typecheck with bindings in Γ, so long as Γ does not introduce any resources whose authority is not already in ε_s. We need Γ to keep track of effects introduced by function arguments. For example, typechecking `f.write` requires a binding for f, but $\lambda f : \{\texttt{File}\}.\ \texttt{f.write}$ does not. Proving the lemma requires us to inductively step into the bodies of functions, at which point we need to keep track of what has been bound — to do this, we permit e to typecheck in a larger environment Γ. We stipulate $\texttt{effects}(\texttt{annot}(\Gamma, \varnothing)) \subseteq \varepsilon_s$ so any effects captured by Γ are not ambient. Note that when $\Gamma = \varnothing$ we have exactly the premises of ε-Import, so when we apply the annotation lemma in the proof of preservation, we choose $\Gamma = \varnothing$. A proof-sketch of the annotation lemma is given below.

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

Case: T-Var. Then $e = x$. If $x \neq y$ use ε-Var and ε-Subsume. Otherwise $x = y$. Then $y : \texttt{erase}(\hat{\tau}) \vdash x : \tau$ implies that $\texttt{erase}(\hat{\tau}) = \tau$. Apply the approximation lemma and simplify to obtain $\hat{\tau} <: \texttt{annot}(\tau, \varepsilon_s)$, then use ε-Subsume to get the result.

Case: T-Resource. Use ε-Resource and ε-Subsume.

Case: T-Abs. Use inversion to get a judgement for the body of the function $\Gamma, y : \texttt{erase}(\hat{\tau}), x : \tau_2 \vdash e_{body} : \tau_3\ \texttt{with}\ \varepsilon_s$. Apply the inductive hypothesis to e_{body} with $\Gamma, x : \tau_2$ as the context in which e_{body} typechecks, noting the premises for the inductive application are satisfied because $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s$ implies $\texttt{effects}(\texttt{annot}(\tau_1, \varnothing) \subseteq \varepsilon_s$. Then use ε-Abs and ε-Subsume.

Case: T-App. Apply the inductive assumption to the subexpressions, then use ε-App and simplify.

Case: T-OperCall. Apply the inductive hypothesis to the receiver and use ε-OperCall. This gives the approximate effects $\varepsilon_s \cup \{\bar{r}.\pi\}$. Consider where the binding for $\{\bar{r}\}$ is in $\hat{\Gamma}, \texttt{annot}(\Gamma, \varepsilon_s), y : \hat{\tau}$ and conclude that $\{\bar{r}.\pi\} \subseteq \varepsilon_s$. □

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

Theorem 3.6 (CC 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* $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B\ \texttt{with}\ \varepsilon_B$, *where* $\hat{e}_B <: \hat{e}_A$ *and* $\varepsilon \cup \varepsilon_B \subseteq \varepsilon_A$, *for some* $\hat{e}_B, \varepsilon, \hat{\tau}_B, \varepsilon_B$.

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

Case: ε-Import. Then $e_A = \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e$. If the reduction rule used was E-Import1 then the result follows by applying the inductive hypothesis to $\hat{e}$. Otherwise $\hat{e}$ is a value and the reduction used was E-Import2. Apply the annotation lemma with $\Gamma = \varnothing$ to get the judgement $\hat{\Gamma}, x : \hat{\tau} \vdash \texttt{annot}(e, \varepsilon_s) : \texttt{annot}(\tau, \varepsilon_s)$ with ε_s. By assumption, $\hat{\Gamma} \vdash \hat{v} : \hat{\tau}$ with $\varnothing$, so the substitution lemma applies, giving $\hat{\Gamma} \vdash [\hat{v}/x]\texttt{annot}(e, \varepsilon) : \texttt{annot}(\tau, \varepsilon_s)$. Then $\varepsilon_B = \varepsilon_s = \varepsilon_A \cup \varepsilon$ and $\tau_A = \tau_B = \texttt{annot}(\tau, \varepsilon_s)$. □

From progress and preservation we can prove the single-step and multi-step soundness theorems for CC. Their proofs are identical to the ones in OC.

Theorem 3.7 (CC Single-step Soundness). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ with ε_A and $\hat{e}_A$ is not a value, then $\hat{e}_A \longrightarrow \hat{e}_B \mid \varepsilon$, where $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B$ with ε_B and $\hat{\tau}_B <: \hat{\tau}_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, for some $\hat{e}_B$, ε, $\hat{\tau}_B$, and ε_B.*

Theorem 3.8 (CC Multi-step Soundness). *If $\hat{\Gamma} \vdash \hat{e}_A : \hat{\tau}_A$ with ε_A and $\hat{e}_A \longrightarrow^* e_B \mid \varepsilon$, then $\hat{\Gamma} \vdash \hat{e}_B : \hat{\tau}_B$ with ε_B, where $\hat{\tau}_B <: \hat{\tau}_A$ and $\varepsilon_B \cup \varepsilon \subseteq \varepsilon_A$, for some $\hat{\tau}_B$, ε_B.*

4 TRANSLATIONS

In this section we develop notation and techniques so our calculi can express the practical examples of the next section. To do this, we show how to encode `unit` and `let` in CC, make some simplifying assumptions, and show how Wyvern-like programs can be translated into CC. With these, we hope to convince the reader that CC adequately captures the properties of capability-safe languages.

4.1 Unit, Let

The `unit` literal is defined as $\texttt{unit} \stackrel{\text{def}}{=} \lambda x : \varnothing.\ x$. It is the same in both annotated and unannotated code. In annotated code, it has the type $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to_{\varnothing} \varnothing$, while in unannotated code it has the type $\texttt{Unit} \stackrel{\text{def}}{=} \varnothing \to \varnothing$. These are technically two separate types, but we will not distinguish between them. Note that `unit` is a value, and because $\varnothing$ is uninhabited (there is no empty resource literal), `unit` cannot be applied to anything. Furthermore, $\vdash \texttt{unit} : \texttt{Unit}$ with $\varnothing$ by ε-Abs, and $\vdash \texttt{unit} : \texttt{Unit}$ by T-Abs. We use `Unit` to represent the absence of information, such as when a function takes no input or returns no value

The expression $\texttt{let}\ x = \hat{e}_1\ \texttt{in}\ \hat{e}_2$ reduces $\hat{e}_1$ to a value $\hat{v}_1$, binds it to the name x in $\hat{e}_2$, and then executes $[\hat{v}_1/x]\hat{e}_2$. If $\hat{\Gamma} \vdash \hat{e}_1 : \hat{\tau}_1$ with ε_1, then $\texttt{let}\ x = \hat{e}_1\ \texttt{in}\ \hat{e}_2 \stackrel{\text{def}}{=} (\lambda x : \hat{\tau}_1.\hat{e}_2)\hat{e}_1$[4]. 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 $\hat{e}_2$. `let` expressions can be typed using ε-App.

4.2 Modules

Wyvern's modules are first-class, desugaring into objects — invoking a module's function is no different from invoking an object's method. Figure 25 shows an example of two modules. The first defines a single operation `tick` that takes an argument file and appends to it; the second is actually a *functor* that takes the file as a module-level argument and uses that in the operation defined. Modules are declared with the `module` keyword, and we use `module def` for functors.

Functors must be instantiated with appropriate arguments in order to produce a usable module. When they are instantiated they are given the capabilities they require. In Figure 25, `Ftor` requests the use of a `File` capability. Figure 26 demonstrates how the two modules above would be used. To prevent infinite regress the `File` must, at some point, be introduced into the program. This happens in the client program. When the program begins execution, the `File` capability is passed into the program from the system environment. The program then

[4]We could also define an unannotated version of `let`, but we only need the annotated version.

```
module Mod: FileTicker

def tick(f: {File}): Unit with {File.append}
  f.append
```

```
module def Ftor(f: {File}): UnitTicker

def tick(): Unit with {File.append}
  f.append
```

Fig. 25. Definition of two modules, the second of which is a functor.

imports modules and instantiates functors with the capabilities they require. If a module is annotated, its function signatures will have effect annotations. For example, in Figure 25, Mod.tick has the File.append annotation, meaning it should typecheck as {File} $\rightarrow_{\{\texttt{File.append}\}}$ Unit. Both Mod and Ftor are annotated.

```
require File

import Mod
instantiate Ftor(File)

Mod.tick(File)
Ftor.tick()
```

Fig. 26. The client program instantiates Mod and Ftor and then invokes tick on each.

Our Wyvern examples are simplified in several ways so they can be expressed in CC. The only objects used are modules. The modules only ever contain one function and the capabilities they require; they have no mutable fields. There are no self-referencing modules or recursive functions. Modules do not reference each other cyclically. These simplifications enable us to model each module as a function. Applying the function will be equivalent to applying the single function defined by the module. A collection of modules is translated into CC as follows. First, a sequence of let-bindings are used to associate the name of a module with the function defined in it, and to associate the name of a functor with a constructor function that, when given the capabilities requested by a functor, will return the function representing a module instance. The constructor for a functor F is called MakeF. If the module does not require any capabilities it takes Unit as its argument. A function is then defined which represents the body of code in the main program. When invoked, this function will instantiate all the functors by invoking their constructors and then will execute the code in from the main program. Finally, the function representing the program is invoked with the primitive capabilities that are passed in from the system environment.

Figure 27 shows how the examples above translate. Lines 1-2 define the module Mod. Lines 4-6 define the constructor for Ftor. It requires a File capability, so the constructor takes {File} as its input type on line 5. The constructor for the main program is defined on lines 8-12, which instantiates Ftor and runs the main program code. Line 14 starts execution by invoking MakeMain with the initial set of capabilities, which in this case is just File.

```
let Mod =
  λf: {File}. f.append in

let MakeFtor =
  λf: {File}.
    λx: Unit. f.append in

let MakeMain =
  λf: {File}.
    let Ftor = (MakeFtor f) in
    let r1 = (Mod f) in
    Ftor unit

MakeMain File
```

Fig. 27. Translation of Mod and Ftor into CC.

When an unannotated module is translated into CC, the translated contents will be encapsulated with an `import` expression. The selected authority on the `import` expression will be that we expect of the unannotated code according to the principle of least authority in the particular example under consideration. For example, if the client only expects the unannotated code to have the File.append effect, the corresponding `import` expression will select {File.append}.

5 APPLICATIONS

In this section we show how the capability-based design of CC can assist in reasoning about the effects and behaviour of a program. We present several scenarios which demonstrate unsafe behaviour or a particular developer story. This takes the form of writing a Wyvern program, translating it to CC using the techniques of the previous section, and then explaining how the rules of CC apply. In discussing these examples, we hope to illustrate where the rules of CC may arise in practice, and convince the reader that they adequately capture the intuitive properties of capability-safe languages like Wyvern.

5.1 Unannotated Client

There is a single primitive capability `File`. A `logger` module possessing this capability exposes a function `log` which incurs `File.write` when executed. The `client` module, possessing the `logger` module, exposes a functino `run` which invokes `logger.log`, incurring `File.write`. While `logger` has been annotated, `client` has not — if `client.run` is executed, what effects might it have? Code for this example is given below.

```
module def logger(f: {File}):Logger

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

```
module def client(logger: Logger)

def run(): Unit =
  logger.log()
```

```
require File

instantiate logger(File)
instantiate client(logger)

client.run()
```

The translation is given below. It first creates two functions, `MakeLogger` and `MakeClient`, which instantiate the `logger` and `client` modules. Lines 1-3 define `MakeLogger`. When given a `File`, it returns a function representing `logger.log`. Lines 5-8 define `MakeClient`. When given a `Logger`, it returns a function representing `client.run`. Lines 10-15 define `MakeMain` which returns a function which, when executed, instantiates all other modules and invokes the code in the body of `Main`. Program execution begins on line 16, where `Main` is given the initial capabilities — which, in this case, is just `File`.

```
1  let MakeLogger =
2    (λf: File.
3      λx: Unit. f.append) in
4
5  let MakeClient =
6    (λlogger: Unit →_{File.append} Unit.
7      import(File.append) l = logger in
8        λx: Unit. l unit) in
9
10 let MakeMain =
11   (λf: File.
12       let loggerModule = MakeLogger f in
13       let clientModule = MakeClient loggerModule in
14       clientModule unit) in
15
16 MakeMain File
```

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

5.2 Unannotated Library

The next example inverts the roles of the last scenario: now, the annotated `client` wants to use the unannotated `logger`. `logger` captures `File` and exposes a single function `log` which incurs the `File.append` effect. `client` has a function `run` which executes `logger.log`, incurring its effects. `client.run` is annotated with ∅, so the implementation of `logger.log` violates its interface.

```
module def logger(f: {File}): Logger

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

```
module def client(logger: Logger)

```

```
def run(): Unit with {File.append} =
  logger.log()
```

```
require File

instantiate logger(File)
instantiate client(logger)

client.run()
```

The translation is given below. On lines 3-4, the unannotated code is wrapped in an `import` expression selecting `{File.append}` as its authority. The implementation of `logger` actually abides by this selected authority, but it has the authority to perform any operation on `File`, so it could, in general, invoke any of them. ε-IMPORT rejects this example because the imported capability has the type `{File}` and `effects({File}) = {File.*}` $\not\subseteq$ `{File.append}`.

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

The only way for this to typecheck would be to annotate `client.run` as having every effect on `File`. This demonstrates how the effect-system of CC approximates unannotated code: it simply considers it as having every effect which could be incurred on those resources in scope, which here is `File.*`.

5.3 Higher-Order Effects

In this scenario, `Main` gains its functionality from a plugin. Plugins might be written by third-parties, in which case we may not be able to view their source code, but still want to reason about the authority they exercise. In this example, `plugin` has access to a `File` capability, but its int erface does not permit it to perform any operations on `File`. It tries to subvert this by wrapping the capability inside a function and passing it to `malicious`, which invokes `File.read` in a higher-order manner in an unannotated context.

```
module malicious

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

```
module plugin
import malicious

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

```
require File
import plugin

plugin.run(File)
```

This example shows how higher-order effects can obfuscate potential security risks. On line 3 of `malicious`, the argument to `log` has type `Unit` → `Unit`. The body of `log` types with the T-rules, which do not approximate effects. It is not clear from inspecting the unannotated code that a `File.read` will be incurred. To realise this requires one to examine the source code of both `plugin` and `malicious`.

A translation is given below. On lines 2-3, the `malicious` code selects its authority as ∅, to be consistent with the annotation on `plugin.run`. This example is rejected by ε-Import. When the unannotated code is annotated with ∅, it has type {`File`} $\rightarrow_{\varnothing}$ `Unit`. The higher-order effects of this type are `File.*`, which is not contained in the selected authority ∅ — hence, ε-Import safely rejects the program.

```
let malicious =
  (import(∅) y=unit in
    λf: Unit → Unit. f()) in

let plugin =
  (λf: {File}.
    malicious(λx:Unit. f.read)) in

let MakeMain =
  (λf: {File}.
    plugin f) in

MakeMain File
```

To get this example to typecheck, the `import` expression has to select {`File.*`} as its authority, and `plugin.run` needs to be annotated with {`File.*`}. In other words, the program would have to be rewritten to explicitly say that plugins can exercise authority over `File`.

5.4 Resource Leak

This is another example which obfuscates an unsafe effect by invoking it in a higher-order manner. The setup is the same, except the function which `plugin` passes to `malicious` now returns `File` when invoked. `malicious` uses this function to obtain `File` and directly invokes `read` upon it, violating the supposed purity of `plugin`.

```
module malicious

def log(f: Unit → File):Unit
  f().read
```

```
module plugin
import malicious

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

```
require File

import plugin

plugin.run(File)
```

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

```
1  let malicious =
2    (import(∅) y=unit in
3      λf: Unit → {File}. f().read) in
4
5  let plugin =
6    (λf: {File}.
7      malicious(λx:Unit. f)) in
8
9  let MakeMain =
10   (λf: {File}.
11     plugin f) in
12
13 MakeMain File
```

6 CONCLUSIONS

We introduced OC, a lambda calculus with primitive capabilities and their effects. OC programs are fully annotated with their effects. Relaxing this requirement, we obtained CC, which allows unannotated code to be nested inside annotated code with a new `import` construct. The capability-safe design of CC allows us to safely infer the effects of unannotated code by inspecting what capabilities are passed into it by its annotated surroundings. Such an approach allows code to be incrementally annotated, giving developers a balance between safety and convenience and alleviating the verbosity that has discouraged widespread adoption of effect systems (Rytz et al. 2012).

More broadly, our results demonstrate that the most basic form of capability-based reasoning—that you can infer what code can do based on what capabilities are passed to it—is not only useful for informal reasoning, but can improve formal reasoning about code by reducing the necessary annotation overhead.

6.1 Related Work

While much related work has already been discussed as part of the presentation, here we cover some additional strands related to capabilities and effects.

Capabilities were introduced by Dennis and Van Horn (1966) to control which processes in an operating system had permission to access which operating system resources. These early ideas were adapted to the programming language setting as the object capability model, exemplified in the work of Mark Miller (2006), which constrains how permissions may proliferate among objects in a distributed system. Maffeis et al. (2010) formalised the notion of a capability-safe language and showed that a subset of Caja (a Javascript implementation) is capability-safe. Miller's model has been applied to more heavyweight systems: Drossopoulou et al. (2007) combined Hoare logic with capabilities to formalise the notion of trust. Capability-safety parallels have been explored in the operating systems literature, where similar restrictions on dynamic loading and resource access (Hunt et al. 2007) enable static, lightweight analyses to enforce privilege separation (Madhavapeddy et al. 2013).

The original effect system by Lucassen and Gifford (1988) was used to determine what expressions could safely execute in parallel. Subsequent applications include determining what functions a program might invoke (Tang 1994) and what regions in memory might be accessed or updated during execution (Talpin and Jouvelot 1994). In these systems, "effects" are performed upon "regions"; in ours, "operations" are performed upon "resources"'. `CC` also distinguishes between unannotated and annotated code: only the latter will type-and-effect-check. Another capability-based effect system is the one by Devriese et al. (2016), who use effect polymorphism and possible world semantics to express behavioural invariants on data structures. `CC` is not as expressive, since it only topographically inspects how capabilities can be passed around a program, but the resulting formalism and theory is much more lightweight.

6.2 Future Work

Our effects model only the use of capabilities which manipulate system resources. This definition could be generalised to track other sorts of effects, such as stateful updates. In our model, resources and operations are fixed throughout runtime; it would be interesting to consider the theory when they can be created and destroyed at runtime.

The current theory contains no effect polymorphism, whereby a function's type is parameterised by a set of effects. The only way for such a function to typecheck in `CC` would be to approximate it as having every effect, in which case all precision has been lost. A polymorphic effect system which considers such a function as having an effect parameterised type could give more meaningful approximations.

Many believe in the value of the object capability model, but we do not fully understand its formal benefits. We hope to extend the ideas in this paper to the point where they might be used in capability-safe languages to help authority-safe design and development. Implementing these ideas in a general-purpose, capability-safe language would do much towards that end.

While we have captured the most obvious and basic form of capability-based reasoning about effects, the ideas here could potentially be useful in other kinds of formal reasoning system. Furthermore, there may be other kinds of reasoning about capabilities—e.g. those being explored by Drossopoulou et al. (2007)—that also provide benefit in a broad set of formal tools.

REFERENCES

Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. 2015. Evaluating the Flexibility of the Java Sandbox. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, New York, NY, USA, 1–10. DOI: http://dx.doi.org/10.1145/2818000.2818003

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155. DOI: http://dx.doi.org/10.1145/365230.365252

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. IEEE European Symposium on Security and Privacy. In *Reasoning about Object Capabilities with Logical Relations and Effect Parametricity.*

Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. IEEE European Symposium on Security and Privacy. In *Computer Security Foundations Symposium.*

Sophia Drossopoulou, James Noble, Mark S. Miller, and Toby Murray. 2007. Reasoning about Risk and Trust in an Open World. In *European Conference on Object-Oriented Programming*. 451–475.

Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 341–354. DOI: http://dx.doi.org/10.1145/1272998.1273032

Joseph R. Kiniry. 2006. *Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application.* Springer Berlin Heidelberg, Berlin, Heidelberg, 288–300. DOI: http://dx.doi.org/10.1007/11818502_16

Darya Kurilova, Alex Potanin, and Jonathan Aldrich. 2016. Modules in Wyvern: Advanced Control over Security and Privacy. In *Proceedings of the Symposium and Bootcamp on the Science of Security (HotSos '16)*. ACM, New York, NY, USA, 68–68. DOI: http://dx.doi.org/10.1145/2898375.2898376

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Mathematically Structured Functional Programming 2014. (March 2014). https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 47–57. DOI: http://dx.doi.org/10.1145/73560.73564

Michael Maass. 2016. *A Theory and Tools for Applying Sandboxes Effectively.* Ph.D. Dissertation. Carnegie Mellon University.

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Not.* 48, 4 (March 2013), 461–472. DOI: http://dx.doi.org/10.1145/2499368.2451167

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 125–140. DOI: http://dx.doi.org/10.1109/SP.2010.16

Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. 2003. *Capability Myths Demolished.* Technical Report.

Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University.

Flemming Nielson and Hanne Riis Nelson. 1999. Type and Effect Systems. *Commun. ACM* (1999), 114–136. DOI: http://dx.doi.org/10.1145/361604.361612

Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2013. Wyvern: A Simple, Typed, and Pure Object-oriented Language. In *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and inHerItance (MASPEGHI '13)*. ACM, New York, NY, USA, 9–16. DOI: http://dx.doi.org/10.1145/2489828.2489830

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 258–282. DOI: http://dx.doi.org/10.1007/978-3-642-31057-7_13

J.-P. Talpin and P. Jouvelot. 1994. The type and effect discipline. *Information and Computation* 111, 2 (1994), 245–296.

Y.-M. Tang. 1994. *Control-Flow Analysis by Effect Systems and Abstract Interpretation.* Ph.D. Dissertation. Ecole des Mines de Paris.