Capabilities: Effects for Free

Aaron Craig[1], Alex Potanin[1][0000−0002−4242−2725], Lindsay Groves[1], and Jonathan Aldrich[2][0000−0003−0631−5591]

[1] School of Engineering and Computer Science, Victoria University of Wellington, NZ
{aaron.craig,alex,lindsay}@ecs.vuw.ac.nz
[2] School of Computer Science, Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract. Object capabilities are increasingly used to reason informally about the properties of secure systems. But 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 to support capability-based reasoning. We demonstrate that capabilities provide a way to reason 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 it relies only on type-checking (not effect-checking), does not require the programmer to add effect annotations within the expression, and does not require the expression to be analysed for its effects. Our result sheds light on the essence of what capabilities provide and suggests ways of integrating lightweight capability-based reasoning into languages.

1 Introduction

Capabilities have been recently gaining attention as a promising mechanism for controlling access to resources, particularly in object-oriented languages and systems [**?**,**?**,**?**,**?**]. A *capability* is an unforgeable token that can be used by its bearer to perform some operation on a resource [**?**]. In a *capability-safe* language, all resources must be accessed through object capabilities, and a resource-access capability must be obtained from an object that already has it: "only connectivity begets connectivity" [**?**]. For example, a `logger` component that provides a logging service would need to be initialised with an object capability providing the ability to append to the log file.

Capability-safe languages prohibit the *ambient authority* [**?**] that is present in non-capability-safe languages. An implementation of a `logger` in Java, for example, does not need to be initialised with a log file capability, as it can simply import the appropriate file-access library and open the log file for appending by itself. But critically, a malicious implementation 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 damage of such malicious components, but recent work has found that Java's sandbox (for instance) is difficult to use and therefore often misused [**?**,**?**].

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. [**?**] 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. [**?**] developed a formalism for reasoning about which components may use a capability and which may influence (perhaps indirectly) the use of a capability, while Devriese et al. [**?**] 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.

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 approaches?

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 [**?**]. Effect systems are a relatively simple formal reasoning approach, which augment type systems with the ability to reason about dynamic effects — and keeping things simple will help to highlight the difference made by capabilities. Effects also 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.

One challenge to the wider adoption of effect systems is their annotation overhead [**?**]. For example, Java's checked exception system, which is a kind of effect system, is often criticised for being cumbersome [**?**]. While effect inference can be used to reduce the annotations required [**?**], 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:

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

Fig. 1. Declaring an effect

Our examples are written in a capability-safe language supporting first-class, object-like modules, similar to *Wyvern* [**?**], in which expressions declare what capabilities they need to execute. In this case, an expression `e` must be passed a function of type `String` $\rightarrow$ `Unit`,[3] which incurs no more than the effect `File.write` when invoked. This function is bound to the name `log` inside `e`.

What can we say about the effects that evaluating `e` will have on resources, such as the file system or network? Because we are in a capability-safe language, `e` has no ambient authority, so the only way it could have any effects is via the `log` function given to it. Since the `log` function is annotated as having no more than the `File.write` effect, this is an upper-bound on the effects of `e`. Note we only required that `e` obeys the rules of capability safety. We did not require it to have effect annotations, and we didn't analyse its structure, as an effect inference would. Also note that `e` might be arbitrarily large, perhaps consisting of an entire program we have downloaded from a source we trust enough to write to a log, but not enough to access any other resources. Thus in this

[3] `Unit` is a singleton type, like `void` in C and Java.

scenario, capabilities can be used to reason "for free" about the effects of a large body of code (`e`), based on a few annotations on the components it imports (`log`).

This example illustrates the central intuition of this paper: in a capability-safe setting, the effects of an unannotated expression can be bounded by the effects latent in the variables that are in scope. In the remainder of this paper, we formalise these ideas in a capability calculus (CC; section 2). Along the way we must generalise this intuition: what if `log` takes a higher-order argument? If `e` evaluates, not to `unit`, but to a function, what can we say about its effects? We then show how CC can model practical situations by encoding a range of Wyvern-like programs section 3). A more thorough discussion, including a proof of soundness is given in an accompanying technical report [**?**].

2 Capability Calculus (CC)

While the current resurgence of interest in capabilities is primarily focused on object-oriented languages, for simplicity our formal definitions build on a typed lambda calculus with a simple notion of capabilities and their operations. CC permits the nesting of unannotated code inside annotated code in a controlled, capability-safe manner using the `import` form from Figure 1. This allows us to reason about unannotated code by inspecting what capabilities are passed into it from its unannotated surroundings.

Allowing effect-annotated and unannotated code to be mixed helps reduce the cognitive overhead on developers, allowing them to prototype in the unannotated sublanguage and incrementally add annotations as they are needed. Reasoning about unannotated code is difficult in general. Figure 2 demonstrates why: `apply` 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 `apply`.

```
1 def apply(f: Unit → Unit):
2   f()
```

Fig. 2. What effects can `apply` incur?

Consider another scenario, where a developer must decide whether or not to use the `logger` functor defined in Figure 3. This functor takes two capabilities as input, `File` and `Socket`.[4] It instantiates an object-like module that has a single, unannotated `log` method with access to these capabilities. The type of this object-like module is `Logger`, which is assumed to be defined elsewhere.

```
1 module def logger(f:{File},s:{Socket}):Logger
2 def log(x: Unit): Unit
3   ...
```

Fig. 3. In a capability-safe setting, `logger` can only exercise authority over the `File` and `Socket` capabilities given to it.

[4] Note that the resource literal is `File`, while the type of the resource literal is {`File`}.

How can we determine what effects will be incurred if `Logger.log` is invoked? One approach is to manually[5] 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, since a `Logger` can only exercise the authority it is explicitly given. In this case, the `logger` functor must be given `File` and `Socket`, so an upper bound on the effects of the `Logger` it instantiates will be the set of all operations on those resources, $\{\texttt{File}.*, \texttt{Socket}.*\}$. Knowing the `Logger` could perform arbitrary reads and writes to `File`, or communicate with `Socket`, the developer decides this implementation cannot be trusted and does not use it.

To model this situation in `CC`, we add a new `import` expression that selects what authority ε_s the unannotated code may exercise. In the above example, the expected least authority of `Logger` is $\{\texttt{File.append}\}$, so that is what the corresponding `import` would select. The type system can then check whether the capabilities being passed into the unannotated code exceed ε_s. If it accepts, then ε_s is a safe upper bound on 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.

2.1 Grammar (`CC`)

The grammar of `CC` has 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 4. Unannotated types τ are built using $\rightarrow$ and sets of resources $\{\bar{r}\}$. An unannotated context Γ maps variables to unannotated types. The syntax for invoking an operation on a resource is $e.\pi$. Resource literals and operations are drawn from fixed sets R (containing, e.g. `File`, `Socket`) and Π (containing, e.g. `write`, `read`).

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

Fig. 4. Unannotated programs and types in `CC`.

Because our focus is on tracking what effects happen, i.e. whether particular operations are invoked on particular resources, we make the following simplifying assumptions: first, any operation may be called on any resource literal; and second, all operations take no inputs and return `unit`.

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

$$
\begin{array}{rll}
\hat{e} ::= & & \textit{labelled exprs}: \\
\mid & x & \textit{variable} \\
\mid & \hat{v} & \textit{value} \\
\mid & \hat{e}\ \hat{e} & \textit{application} \\
\mid & \hat{e}.\pi & \textit{operation} \\
\mid & \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e & \textit{import} \\
\\
\hat{v} ::= & & \textit{labelled values}: \\
\mid & r & \textit{resource literal} \\
\mid & \lambda x : \hat{\tau}.\hat{e} & \textit{abstraction}
\end{array}
\qquad
\begin{array}{rll}
\hat{\tau} ::= & & \textit{annotated types}: \\
\mid & \{\bar{r}\} & \textit{resource set} \\
\mid & \hat{\tau} \to_\varepsilon \hat{\tau} & \textit{function} \\
\\
\hat{\Gamma} ::= & & \textit{annotated type ctx}: \\
\mid & \varnothing & \textit{empty ctx.} \\
\mid & \hat{\Gamma}, x : \hat{\tau} & \textit{binding} \\
\\
\varepsilon ::= & & \textit{effects}: \\
\mid & \{\overline{r.\pi}\} & \textit{effect set}
\end{array}
$$

Fig. 5. Annotated programs and types in CC.

Rules for annotated programs and their types are shown in Figure 5. The first main difference is that the $\to_\varepsilon$ type constructor has a subscript ε, which is a set of effects that functions of that type may incur. The other main difference is the new expression form, $\texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e$, where e is some unannotated code and $\hat{e}$ is a capability being passed to it; we call $\hat{e}$ an import. For simplicity, we assume there is only ever one import. Note the definition not only allows resource literals to be imported, but also effectful functions. Inside e, $\hat{e}$ is bound to the variable x. ε_s is the maximum authority that e is allowed to exercise (its "selected authority"). For example, suppose an unannotated `Logger`, which requires `File`, is expected to only `append` to a file, but has an implementation which writes. This would be the expression $\texttt{import}(\texttt{File.append})\ x = \texttt{File}\ \texttt{in}\ \lambda y : \texttt{Unit}.\ \texttt{x.write}$. The `import` expression 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 effects of unannotated code. For the rest of our discussion of CC, we will only be interested in unannotated code when it is encapsulated by an `import` expression.

Capability safety prohibits ambient authority. CC meets this requirement by forbidding the use of resource literals directly inside an `import` expression (though they can still be passed in as a capability via the binding variable x). We could have enforced this syntactically, but we choose to do it using the typing rule for `import` in section 2.3.

2.2 Semantics (CC)

The rules for CC are natural extensions of the simply-typed lambda calculus, so for brevity we only give the rules for `import` (see Figure 6). Reductions are defined on annotated expressions, using the notation $\hat{e} \longrightarrow \hat{e}' \mid \varepsilon'$, which means that $\hat{e}$ is reduced to $\hat{e}'$ in a single step, incurring the set of effects ε'. To execute the unannotated code inside an `import` expression, we recursively annotate its components with the selected authority ε_s. While it is meaningful to execute unannotated code, we only care about it inside `import` expressions, so do not bother to give rules for this.

E-IMPORT1 reduces the capability being imported. When it has been reduced to a value $\hat{v}$, E-IMPORT2 annotates e with the selected authority ε — this is $\texttt{annot}(e, \varepsilon)$ — and substitutes the import $\hat{v}$ for its name x in e — this is $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$.

$\texttt{annot}(e, \varepsilon)$ is the expression obtained by recursively annotating the parts of e with the set of effects ε. A definition is given in Figure 7, with versions defined on expressions and types. Later we will need to annotate contexts, so the definition is given here.

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

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

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

Fig. 6. New single-step reductions in CC.

Note 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.

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

$$\begin{aligned}
&\texttt{annot} :: \tau \times \varepsilon \to \hat{\tau}\\
&\texttt{annot}(\{\bar{r}\}, _) = \{\bar{r}\}\\
&\texttt{annot}(\tau_1 \to \tau_2, \varepsilon) = \texttt{annot}(\tau_1, \varepsilon) \to_\varepsilon \texttt{annot}(\tau_2, \varepsilon).
\end{aligned}$$

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

Fig. 7. Definition of annot.

2.3 Static Rules (CC)

Terms 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 involving them only ascribe a type to an expression, with the form $\Gamma \vdash e : \tau$. Subtyping judgements have the form $\tau <: \tau$. Because these rules are essentially those of the simply-typed lambda calculus, we do not list them here.

Judgements involving annotated terms have the form $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon$, meaning that when $\hat{e}$ is evaluated, it reduces to a value of type $\hat{\tau}$, incurring no more than the effects in ε. Most of the rules are analogous to those of the simply-typed lambda calculus; these ones are given in Figure 8. Note that the rule for typing an operation call, ε-OPERCALL, types the expression as Unit, following our simplifying assumption that all operations return Unit.

There is one rule left, for typing import. Since it is a complicated rule, we will start with a simplified (but incorrect) version, and spend the rest of the section building up to the final version.

To begin, typing $\texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e$ in a context $\hat{\Gamma}$ requires us to know that $\hat{e}$ is well-typed, so we add the premise $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1$. e is only allowed to use what authority has been explicitly given to it (i.e. the capability $\hat{e}$, bound to x). To ensure this, we require that e can be typechecked using only one binding, $x : \hat{\tau}$, which binds x to the type of the capability being imported. Typing e in this restricted environment means it cannot use any other capabilities, thus prohibiting the exercise of ambient authority.

There is a problem though: e is unannotated, while $\hat{\tau}$ is annotated, and there is no rule for typechecking unannotated code in an annotated context. To get around this, we

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

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

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

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

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

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

Fig. 8. Type-and-effect and subtyping judgements in CC.

define a function `erase` in Figure 9, which removes the annotations from a type. We can then add $x : \texttt{erase}(\hat{\tau}) \vdash e : \tau$ as a premise.

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

Fig. 9. Definition of `erase`.

The first version of ε-IMPORT is given in Figure 10. Since $\texttt{import}(\varepsilon_s)\ x = \hat{v}\ \texttt{in}\ e$ reduces to $[\hat{v}/x]\texttt{annot}(e, \varepsilon_s)$ by E-IMPORT2, its ascribed type is $\texttt{annot}(\tau, \varepsilon)$, which is the type of the unannotated code e, annotated with its selected authority ε_s. The effects of reducing the `import` are $\varepsilon_1 \cup \varepsilon_s$ — the former happens when the imported capability is reduced to a value, while the latter happens when the body of the `import` expression is annotated and executed.

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon_s \cup \varepsilon_1}\ (\varepsilon\text{-IMPORT1-BAD})$$

Fig. 10. A first (incorrect) rule for type-and-effect checking `import` expressions.

This first rule is incomplete, since any capability can be passed to the unannotated code e, even if it has effects that weren't declared in ε_s. To avoid this, we define a function `effects`, which collects the set of effects that an (annotated) type captures. For example, $\{\texttt{File}\}$ captures every operation on `File`, so $\texttt{effects}(\{\texttt{File}\}) = \{\texttt{File}.*\}$. A first (but not yet correct) definition of this is given in Figure 11. We then add the premise $\texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s$, which restricts imported capabilities to only those with effects selected in ε_s. The updated rule for typing `import` is given in Figure 12.

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

Fig. 11. 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. 12. A second (still incorrect) rule for type-and-effect checking `import` expressions.

There are still issues with this second rule, as the annotations on one import can be broken by another import. To illustrate, consider Figure 13 where two[6] capabilities are imported. This program imports a function `go` which, when given a `Unit` $\to_\varnothing$ `Unit` function with no effects, will execute it. The other import is `File`. The unannotated code creates a `Unit` $\to$ `Unit` function which writes to `File` and passes it to `go`, which subsequently incurs `File.write`.

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

Fig. 13. 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` $\to$ `unit`, the unannotated world accepts it. Although the unannotated code is allowed to incur this effect, since its selected authority is $\{\texttt{File}.*\}$, this nonetheless violates the type signature of `go`. We want to prevent this.

If `go` had the type `Unit` $\to_{\{\texttt{File.write}\}}$ `Unit`, Figure 13 would be safely rejected. However, a modified program 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. To ensure this is the case, we shall require imported capabilities to have the authority to incur every effect in ε_s. To achieve greater control in how we say this, we split the definitions of `effects` into two separate functions, `effects` and `ho-effects`. The latter is for higher-order effects, which are those effects not captured directly in the function body, but rather are possible because of what is passed into the function 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 $r.\pi$, but need to be given the capability (as a function argument) by someone else to do so, then $r.\pi \in \texttt{ho-effects}(\hat{\tau})$. Definitions are given in Figure 14.

Both `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

[6] 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.

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

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

Fig. 14. Effect functions (corrected).

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 $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau})$, which requires any higher-order effect of the imported capabilities to be declared in ε_s. Put another way, the imported capabilities must be expecting every effect they could be given by the unannotated code (which is at most ε_s). The counterexample from Figure 13 is now rejected, because $\texttt{ho-effects}((\texttt{Unit} \to_\varnothing \texttt{Unit}) \to_\varnothing \texttt{Unit}) = \varnothing$, but $\texttt{effects}(\texttt{File}) = \{\texttt{File}.*\} \not\subseteq \varnothing$.

This is *still* not sufficient! Consider $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2)$. 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 $r.\pi$. If $\hat{\tau}_2$ is a function, unannotated code could violate its annotations by passing it a capability for $r.\pi$, even though $r.\pi$ is not a higher-order effect of $\hat{\tau}_2$.

The cause of this 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 15 defines this: safe is a distributive version of $\varepsilon_s \subseteq \texttt{effects}(\hat{\tau})$ and ho-safe is a distributive version of $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau})$. An amended version of ε-IMPORT is given in Figure 16, with a new premise $\texttt{ho-safe}(\hat{\tau}, \varepsilon_s)$, capturing the notion that imported capabilities must be expecting the effects they could be passed by the unannotated code (which is at most ε_s).

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

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

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

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

Fig. 15. Safety judgements in CC.

The premises so far restrict what authority can be selected by unannotated code, but consider the example $\hat{e} = \texttt{import}(\varnothing)\ x = \texttt{unit in}\ \lambda f : \texttt{File}.\ \texttt{f.write}$. The unannotated code selects no capabilities and returns a function which takes File and incurs File.write. This satisfies the premises in ε-IMPORT3, but its type would be the pure function $\{\texttt{File}\} \to_\varnothing \texttt{Unit}$.

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \texttt{with}\ \varepsilon_1 \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s \quad \texttt{ho-safe}(\hat{\tau}, \varepsilon_s) \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \texttt{annot}(\tau, \varepsilon_s)\ \texttt{with}\ \varepsilon \cup \varepsilon_1}\ (\varepsilon\text{-IMPORT3-BAD})$$

Fig. 16. A third (still incorrect) rule for type-and-effect checking `import` expressions.

Speaking more generally, suppose the unannotated code evaluates to a function of type f, which is annotated to $\texttt{annot}(f, \varepsilon_s)$. Suppose $\texttt{annot}(f, \varepsilon_s)$ is invoked at a later point, back in the annotated world, incurring $r.\pi$. 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, as an argument, a capability to do $r.\pi$, 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 our calculus. But if the argument is a resource literal r, then $\texttt{annot}(f, \varepsilon_s)$ could exercise $r.\pi$ without declaring it in ε_s — this we do not yet account for.

To make ε_s contain every effect captured by resources passed into $\texttt{annot}(f, \varepsilon_s)$ as arguments, we inspect f for resource types. For example, if the unannotated code evaluates to a function of type $\{\texttt{File}\} \to \texttt{Unit}$, we need $\{\texttt{File}.*\} \in \varepsilon_s$. To do this, we add a new premise $\texttt{ho-effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s$. Because `ho-effects` is only defined on annotated types, we first annotate τ with $\varnothing$, and since we are only inspecting the resources passed into f as arguments, our choice of annotation doesn't matter.

Now we can 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 17. 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 $\texttt{import}(\texttt{File.append})\ f = \texttt{File}\ \texttt{in}\ \lambda x : \texttt{Unit}.\ e$. The imported capability is $f = \texttt{File}$, which has type $\{\texttt{File}\}$, and $\texttt{effects}(\{\texttt{File}\}) = \{\texttt{File}.*\} \not\subseteq \{\texttt{File.append}\}$, so this example safely rejects: `Logger.log` has authority to do anything with `File`, and its implementation e might be violating its stipulated least authority $\{\texttt{File.append}\}$.

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

Fig. 17. The final rule for typing imports.

3 Applications

In this section, we examine a number of scenarios to show how capabilities can help developers reason about the effects and behaviour of code. In each story we will discuss some Wyvern code before translating it to CC and explaining how its rules apply. By

doing this, we hope to convince the reader of the benefits of capability-based reasoning, and that CC captures the intuitive properties of capability-safe languages like Wyvern.

3.1 Unannotated Client

A `logger` module, when given `File`, exposes a `log` function which incurs the effect `File.append`. The `client` module, possessing the `logger` module, exposes an unannotated function `run`. 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()
```

A translation into CC is given below. Lines 1-3 and 5-8 define `MakeLogger` and `MakeClient`, which instantiate the `logger` and `client` modules respectively (represented as functions). Lines 10-14 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 (just `File` in this case).

```
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 matches the effects of `logger`, i.e. `effects`(`Unit` $\rightarrow_{\{\texttt{File.append}\}}$ `Unit`) = {`File.append`}. The unannotated code typechecks by ε-IMPORT, approximating its effects as {`File.append`}.

3.2 Unannotated Library

The next example inverts the roles of the last scenario. Now, the annotated `client` wants to use the unannotated `logger`, which captures `File` and exposes a single function `log`, which incurs the `File.append` effect. The implementation of `client.run` executes `logger.log`; it is annotated with $\varnothing$, so this 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, but since it captures `File` it could, in general, do anything to `File`; therefore, ε-IMPORT rejects this example. Formally, the imported capability has the type $\{$`File`$\}$, but $\texttt{effects}(\{\texttt{File}\}) = \{\texttt{File}.*\} \not\subseteq \{\texttt{File.append}\}$. The only way for this to typecheck would be to annotate `client.run` as having every effect on `File`.

```
let MakeLogger =
  (λf: File.
    import(File.append) f = f in
      λx: Unit. f.append) in

let MakeClient =
  (λlogger: Logger.
    λx: Unit. logger unit) in

let MakeMain =
  (λf: File.
    let loggerModule = MakeLogger f in
    let clientModule = MakeClient loggerModule in
    clientModule unit) in

MakeMain File
```

3.3 Higher-Order Effects

Here, `Main` gains its functionality from a plugin. Plugins might be written by third-parties, so 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 `File`, but its interface does not permit it to perform any operations on `File`. It tries to subvert this by wrapping `File` 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`. ε-IMPORT safely rejects this: when the unannotated code is annotated with ∅, it has type $\{\texttt{File}\} \to_{\varnothing} \texttt{Unit}$, but the higher-order effects of this type are $\{\texttt{File}.*\}$, which are not contained in the selected authority ∅.

```
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 program would have to be rewritten to explicitly say that plugins can exercise arbitrary authority over `File`, by changing the selected authority of `import` and the annotation on `plugin.run`.

3.4 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 declared 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 on lines 5-6. It has selected authority is $\varnothing$, to be consistent with the annotation on `plugin`. Nothing is being imported, so the `import` binds `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 $\tau = (\texttt{Unit} \to \{\texttt{File}\}) \to \texttt{Unit}$. Then $\texttt{annot}(\tau, \varepsilon) = (\texttt{Unit} \to_{\varnothing} \{\texttt{File}\}) \to_{\varnothing} \texttt{Unit}$ and $\texttt{ho-effects}(\texttt{annot}(\tau, \varepsilon)) = \{\texttt{File}.*\}$. Thus, the premise cannot be satisfied and the example is safely rejected.

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

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

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

MakeMain File
```

4 Conclusions

We introduced CC, a lambda calculus with a simple notion of resources and their operations, which allows unannotated code to be nested inside annotated code with a new `import` construct. Its capability-safe design enables us to safely reason about 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, alleviating the verbosity that has discouraged widespread adoption of effect systems [**?**].

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.

4.1 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 [**?**] to control which processes had permission to access which resources in an operating system. These ideas were adapted to the programming language setting, particularly by Mark Miller [**?**], whose object-capability model constrains how permissions may proliferate among objects in a distributed system. [**?**] formalised the notion of a capability-safe language and showed that a subset of Caja (a Javascript implementation) is capability-safe. Miller's object-capability model has been applied to more heavyweight systems, such as [**?**], which formalises the notion of trust in a Hoare logic. Capability-safety parallels have been explored in the operating systems literature, where similar restrictions on dynamic loading and resource access [**?**] enable static, lightweight analyses to enforce privilege separation [**?**].

The original effect system by [**?**] was used to determine what expressions could safely execute in parallel. Subsequent applications include determining what functions a program might invoke [**?**] and what regions in memory might be accessed or updated during execution [**?**]. 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 [**?**], who use effect polymorphism and possible world semantics to express behavioural invariants on data structures. `CC` is not as expressive, since it only inspects how capabilities are passed around a program, but the resulting formalism and theory is much more lightweight. Ongoing work with the Wyvern programming language includes an effect system which partially builds on ideas from this paper [**?**].

4.2 Future Work

Our system only models capabilities which manipulate system resources. This definition could be generalised to track other sorts of effects, such as stateful updates. Resources and their operations are fixed throughout runtime, but we could imagine them being created and destroyed at runtime. Finally, other future work could incorporate polymorphic types and effects.