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. 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 it relies only on type-checking (not effect-checking); does not require the programmer to add effect annotations within the expression; nor does it 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 new attention as a promising mechanism for controlling access to resources, particularly in object-oriented languages and systems [15, 6, 5, 4]. A *capability* is an unforgeable token that can be used by its bearer to perform some operation on a resource [3]. 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" [15]. 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 thus prohibit the *ambient authority* [16] that is present in non-capability-safe languages. An implementation of a `logger` in 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 [2, 12].

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 sheds some light on this question by presenting a logic that formalizes capability-based reasoning about trust between objects [6].

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 [5]. 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 [4] . 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 [17]. Effect systems are a relatively simple formal reasoning approach, and keeping things simple will help to highlight the difference made by capabilities. Finally, 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 [18]. For example, Java's checked exception system is a kind of effect system, and is often criticised for being cumbersome [8]. Effect inference can be used to reduce the annotations required [10], 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 language called the capability calculus `CC` (Section 2). Although 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. One can reason about the effects of the unannotated code 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 that supports first-class, object-like modules, similar to *Wyvern* [9]. We give several examples 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 an accompanying technical report [1].

2 Capability Calculus (`CC`)

Allowing a mix of annotated [with effects] 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 1 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. 1. What effects can `someMethod` incur?

The key insight of our paper is that 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 2. It must be instantiated with two capabilities, `File` and `Socket`, and provides an unannotated function `log`.

What effects will be incurred if `Logger.log` is invoked? One approach is to manually[3] 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

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

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

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

Fig. 2. In a capability-safe setting, `logger` can only exercise authority over the `File` and `Socket` capabilities given to it.

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 $\{\texttt{File}.*, \texttt{Socket}.*\}$ 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 authority ε 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 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.

2.1 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 3. Unannotated types τ are built using $\rightarrow$ and sets of resources $\{\bar{r}\}$. An unannotated context Γ maps variables to unannotated types. Rules for annotated programs and their types are in Figure 4.

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

Fig. 3. Unannotated 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

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

Fig. 4. Annotated programs and types in CC.

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 $\texttt{import}(\texttt{File.append})\ x = \texttt{File in}\ \lambda y : \texttt{Unit. x.write}$.

Observe that `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.

2.2 Semantics (CC)

Reductions are defined on annotated expressions and are natural reduction rules from extended lambda calculus. We omit them for brevity. 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 additional rules for reducing `import` expressions, given in Figure 5: 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 $[\hat{v}/x]\texttt{annot}(e, \varepsilon)$.

$\texttt{annot}(e, \varepsilon)$ produces the expression obtained by recursively annotating the parts of e with the set of effects ε. A definition is given in Figure 6. 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.

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

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

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

Fig. 5. New single-step reductions in CC.

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

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

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

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

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

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

Fig. 6. Definition of `annot`.

2.3 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 7.

The annotated static rules are effectively the same only involving the annotated expressions. The interesting rule is ε-IMPORT, given in Figure 16, 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 $\hat{\Gamma} \vdash \hat{e} : \hat{\tau}\ \mathtt{with}\ \varepsilon_1$. 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 8 which removes the annotations from a type. We then add $x : \mathtt{erase}(\hat{\tau}) \vdash e : \tau$ as a premise.

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

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

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

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

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

Fig. 7. (Sub)typing judgements for the unannotated sublanguage of CC

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

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

Fig. 8. Definition of `erase`.

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.

The first version of ε-IMPORT is given in Figure 9. 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. 9. 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 10. 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 11.

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

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

Fig. 10. 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. 11. 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 12 where two[4] 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`.

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

Fig. 12. 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 $\texttt{unit} \to \texttt{unit}$, the unannotated world accepts it. The approximation is actually safe at the top-level, because the `import` selects $\{\texttt{File}.*\}$, 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 $\texttt{Unit} \to_{\{\texttt{File.write}\}} \texttt{Unit}$ 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 $\texttt{Unit} \to_{\{\texttt{File}.*\}} \texttt{Unit}$, 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 13.

[4] 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. 13. Effect functions (corrected).

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 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 12 is now rejected, because $\texttt{ho-effects}((\texttt{Unit} \to_\varnothing \texttt{Unit}) \to_\varnothing \texttt{Unit}) = \varnothing$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$. However, this is *still* not sufficient! Consider $\varepsilon_s \subseteq \texttt{ho-effects}(\hat{\tau}_1 \to_{\varepsilon'} \hat{\tau}_2)$. 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 14 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. 14. Safety judgements in CC.

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

Fig. 15. A third (still incorrect) rule for type-and-effect checking `import` expressions.

An amended version of ε-IMPORT is given in Figure 15. 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. 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 $\hat{e} = \texttt{import}(\varnothing)\ x = \texttt{unit in } \lambda f : \texttt{File}.\ \texttt{f.write}$. 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 $\{\texttt{File}\} \rightarrow_{\varnothing} \texttt{Unit}$: 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 our calculus (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 $\{\texttt{File}\} \rightarrow \texttt{Unit}$, 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 $x : \texttt{Unit} \vdash \lambda f : \{\texttt{File}\}.\ \texttt{f.write} : \{\texttt{File}\} \rightarrow \texttt{Unit}$. Its higher-order effects are $\texttt{ho-effects}(\texttt{annot}(\{\texttt{File}\} \rightarrow \texttt{Unit}, \varnothing)) = \{\texttt{File}.*\}$, but $\{\texttt{File}.*\} \not\subseteq \varnothing$, so the example is safely rejected.

The final version of ε-IMPORT is given in Figure 16. 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 $\texttt{Unit} \rightarrow \texttt{Unit}$ and implementation e. The expected least authority of `Logger` is $\{\texttt{File.append}\}$, so its corresponding `import` expression would be $\texttt{import(File.append)}\ f = \texttt{File in } \lambda x : \texttt{Unit}.\ e$. 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 $\{\texttt{File.append}\}$.

2.4 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.

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

Fig. 16. The final rule for typing imports.

Theorem 1 (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* $\hat{e}_B, \varepsilon, \hat{\tau}_B, \varepsilon_B$.

Due to small page limit we refer the interested readers to the Technical Report with complete proofs including multi-step soundness for the system presented here [1].

3 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, 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.

3.1 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: 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 $\{\texttt{File.append}\}$ as its authority. This is exactly the effects of the logger, i.e. $\texttt{effects}(\texttt{Unit} \rightarrow_{\{\texttt{File.append}\}} \texttt{Unit}) = \{\texttt{File.append}\}$. 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 $\{\texttt{File.append}\}$.

3.2 Unannotated Library

The next example inverts the roles of the last scenario: now, the annotated `client` wants to use the unannotated `logger`. `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 $\varnothing$, 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 $\{\texttt{File.append}\}$ 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 $\{\texttt{File}\}$ and $\texttt{effects}(\{\texttt{File}\}) = \{\texttt{File}.*\} \nsubseteq \{\texttt{File.append}\}$.

```
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 $\texttt{File}.*$.

3.3 Higher-Order Effects

Here `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 interface 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 $\texttt{Unit} \to \texttt{Unit}$. 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 $\varnothing$, to be consistent with the annotation on `plugin.run`. This example is rejected by ε-IMPORT. When the unannotated code is annotated with $\varnothing$, it has type $\{\texttt{File}\} \to_{\varnothing} \texttt{Unit}$. The higher-order effects of this type are $\texttt{File}.*$, which is not contained in the selected authority $\varnothing$ — hence, ε-IMPORT safely rejects the program.

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

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

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 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 $\varnothing$, 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 $\varepsilon = \mathtt{effects}(\hat{\tau}) \cup \mathtt{ho\text{-}effects}(\mathtt{annot}(\tau, \varepsilon))$ is not satisfied. In this case, $\varepsilon = \varnothing$ and $\tau = (\mathtt{Unit} \to \{\mathtt{File}\}) \to \mathtt{Unit}$. Then $\mathtt{annot}(\tau, \varepsilon) = (\mathtt{Unit} \to_{\varnothing} \{\mathtt{File}\}) \to_{\varnothing} \mathtt{Unit}$ and $\mathtt{ho\text{-}effects}(\mathtt{annot}(\tau, \varepsilon)) = \{\mathtt{File}.*\}$. 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
```

4 Conclusions

We introduced CC, a lambda calculus with primitive capabilities and their effects, 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 [18].

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 [3] 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 [16], which constrains how permissions may proliferate among objects in a distributed system. [14] 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: [6] 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 [7] enable static, lightweight analyses to enforce privilege separation [13].

The original effect system by [11] was used to determine what expressions could safely execute in parallel. Subsequent applications include determining what functions a program might invoke [20] and what regions in memory might be accessed or updated during execution [19]. 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 [4], 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.

4.2 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.

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 such as *Wyvern* 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 systems.

References

1. Aaron Craig, Alex Potanin, L.G.J.A.: Capabilities: Effects for free. Tech. rep., VUW (2018), https://ecs.victoria.ac.nz/Main/TechnicalReportSeries
2. Coker, Z., Maass, M., Ding, T., Le Goues, C., Sunshine, J.: Evaluating the flexibility of the java sandbox. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 1–10. ACSAC 2015, USA (2015).
3. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (Mar 1966).
4. Devriese, D., Birkedal, L., Piessens, F.: Ieee european symposium on security and privacy. In: Reasoning about Object Capabilities with Logical Relations and Effect Parametricity (2016)
5. Dimoulas, C., Moore, S., Askarov, A., Chong, S.: IEEE european symposium on security and privacy. In: Computer Security Foundations Symposium (2014)
6. Drossopoulou, S., Noble, J., Miller, M.S., Murray, T.: Reasoning about risk and trust in an open world. In: ECOOP. pp. 451–475 (2007)
7. Hunt, G., Aiken, M., Fähndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., Wobber, T.: Sealing OS processes to improve dependability and safety. SIGOPS OS Rev. **41**(3), 341–354 (Mar 2007).
8. Kiniry, J.R.: Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application, pp. 288–300. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
9. Kurilova, D., Potanin, A., Aldrich, J.: Modules in wyvern: Advanced control over security and privacy. In: Proceedings of the Symposium and Bootcamp on the Science of Security. pp. 68–68. HotSos '16, ACM (2016).
10. Leijen, D.: Koka: Programming with row polymorphic effect types. In: Mathematically Structured Functional Programming 2014. EPTCS (March 2014).
11. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL. pp. 47–57. POPL '88, USA (1988).
12. Maass, M.: A Theory and Tools for Applying Sandboxes Effectively. Ph.D. thesis, Carnegie Mellon University (2016)
13. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: Library operating systems for the cloud. SIGPLAN Not. **48**(4), 461–472 (Mar 2013).
14. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 125–140. SP '10, IEEE Computer Society.
15. Miller, M., Yee, K.P., Shapiro, J.: Capability myths demolished. Tech. rep. (2003)
16. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Johns Hopkins University (2006)
17. Nielson, F., Nelson, H.R.: Type and Effect Systems. pp. 114–136 (1999).
18. Rytz, L., Odersky, M., Haller, P.: Lightweight polymorphic effects. In: ECOOP. pp. 258–282. (2012)
19. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Information and Computation **111**(2), 245–296 (1994)
20. Tang, Y.M.: Control-Flow Analysis by Effect Systems and Abstract Interpretation. Ph.D. thesis, Ecole des Mines de Paris (1994)