Capabilities and Effects

Aaron Craig, Alex Potanin, Lindsay Groves
ECS, VUW, NZ
alex@ecs.vuw.ac.nz

Jonathan Aldrich
ISR, CMU, USA
jonathan.aldrich@cs.cmu.edu

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

ACM Reference Format:
Aaron Craig, Alex Potanin, Lindsay Groves and Jonathan Aldrich. 2017. Capabilities and Effects. In *Proceedings of ACM SIGPLAN OCAP Workshop (OCAP'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/

1 Introduction

Capability-safe languages prohibit the *ambient authority* [5] that is present in non-capability-safe languages. An implementation of a `logger` in OCaml or Java, for example, does not need to be passed a capability at initialisation time; it can simply import the appropriate file-access library and open the log file for appending itself. Critically, a malicious implementation of such a component could also delete the log, read from another file, or exfiltrate logging information over the network. Other mechanisms such as sandboxing can be used to limit the effects of such malicious components, but recent work has found that Java's sandbox (for example) is difficult to use and is therefore often misused [1].

If capabilities are useful for *informal* reasoning, shouldn't they also aid in *formal* reasoning? Work by Drossopoulou et al. [4] 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. [3] developed a formalism for reasoning about which components may use a capability and which may influence (perhaps indirectly) the use of a capability. Devriese et al. [2] 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?

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 [6] — a relatively simple formal reasoning approach that can make the difference made by capabilities more obvious. Furthermore, effects have an intuitive link to capabilities: in a system that uses capabilities to protect resources, an expression can only have an effect on a resource if it is given a capability to do so.

How could capabilities help with effects? One challenge to the wider adoption of effect systems is their annotation overhead [7]. Effect inference can be used to reduce the annotations required, 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:

```
1 import log : String -> Unit with {File.write} in
2 e
```

In a capability-safe language, what can we infer about the effects on resources (e.g. the file system or network) when evaluating the unannotated code `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 from its annotated surroundings. 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. `e` could also 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 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?

2 Unannotated Client

Consider the following example. There is a single primitive capability `File`, exposing some operations to a system

OCAP'17, 2017, Vancouver, Canada
2017.

resource. 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 function `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?

```
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()
```

Our proposal translates the Wyvern code above to a set of simple nested functions following Newspeak-style object capabilities, as shown below. The first two functions, `MakeLogger` and `MakeClient`, 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 10-14 define `MakeMain` which returns a function that, when executed, instantiates all other modules and executes the code in the body of `Main`. Program execution begins on line 16, where the initial capabilities are passed into `Main` — in this case, just `File`.

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

On line 7, an import expression *selects* the authority {`File.append`} for its unannotated body on line 8. This is a well-typed example, as the body can be typechecked using only those free variables imported, and the authority passed in via the `logger` function does not exceed {`File.append`}. It is therefore safe to assume that {`File.append`} is an upper-bound on the effects incurred from evaluating the body; indeed, a lambda abstraction has no effect, though it could be used to incur one later. But since it was defined inside the import expression (or passed into it), it must also have the effects contained in {`File.append`}, so we can effect-check it as such without having to annotate or infer its parts.

3 Polymorphic Effects

The previous example shows how capability safety can be used to infer the effects in unannotated code by inspecting the capabilites we pass into it. We saw an example where functions with a fixed set of effects were imported, but the same reasoning also applies to types which are *polymorphic* over a set of effects, such as the function below, which is polymorphic over the {`File.write`, `Socket.write`} effects. After fixing a particular subset Φ of these effects, it asks for a function with those effects and then incurs them.

```
polywriter =
  λΦ ⊆ {File.write, Socket.write}.
    (λf: Unit →Φ Unit. f unit)
```

If a piece of unannotated code were given a `polywriter`, it would be safe to approximate its effects as the polymorphic upper bound {`File.write`, `Socket.write`}. But we can do better: if no capability for `Socket.write` is passed in with the `polywriter`, then although it could theoretically accept functions which incur `Socket.write`, it will never be able to obtain one. The example below shows such a situation:

```
import({File.write})
  pw = polywriter
  fw = (λf: Unit. File.write)
in
  e
```

Since only a file-writing capability is passed in with `polywriter`, it can never be made to incur `Socket.write`, so a better approximation of e would be {`File.write`}. While a full exploration of capability rules for effect polymorphism is still being finalised, the discussion above suggests the potential to extend the ideas here to a polymorphic setting.

References

[1] Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. 2015. Evaluating the Flexibility of the Java Sandbox *(ACSAC 2015)*. ACM, 1–10. https://doi.org/10.1145/2818000.2818003

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

[3] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative policies for capability control. In *Computer Security Foundations Symposium*.

[4] Sophia Drossopoulou, James Noble, Toby Murray, and Mark S. Miller. 2015. *Reasoning about Risk and Trust in an Open World*. Technical Report. VUW. http://ecs.victoria.ac.nz/foswiki/pub/Main/TechnicalReportSeries/ECSTR15-08.pdf

[5] Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University.
[6] Flemming Nielson and Hanne Riis Nelson. 1999. Type and Effect Systems. *Commun. ACM*, 114–136. https://doi.org/10.1145/361604.361612
[7] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 258–282.