Abstracting Resource Effects

Valerie Zhao
Wellesley College, United States
vzhao@wellesley.edu

Abstract

In developing secure programs, reasoning about effects on resources can be hindered by obscure changes in program state. Effect systems mitigate this difficulty by describing these changes, but may require large amounts of low-level effect annotations, and create obstacles for reasoning. To reduce overhead, we propose an effect system, with a focus on system resources, that supports effect abstraction. It is being implemented in Wyvern, a capability-safe language.

CCS Concepts • **Software and its engineering → Language features**;

Keywords Effect system, system resource, abstraction

ACM Reference Format:
Valerie Zhao. 2017. Abstracting Resource Effects. In *Proceedings of 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3135932.3135946

1 Introduction

Developing secure programs requires reasoning about effects a computation can have on protected resources, but informal code inspection for this objective can be difficult. Understanding a method's full effect may require examining bodies of several nested method calls, as a computation could involve interleaving operations and hidden changes to program state. One remedy is the effect system, in which effects express changes in program state during method calls, and method annotations of expected effects enforce them while reducing need for in-depth analyses [3, 5, 7, 8, 12]. Yet, effects propagate through method calls, therefore backtracking through calls to understand them may still be needed. Specificity and utility must also be calibrated when defining effects. To optimize program reasoning and combat annotation overhead, we propose an abstraction-based effect system.

SPLASH Companion'17, October 22–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5514-8/17/10…$15.00
https://doi.org/10.1145/3135932.3135946

2 Proposed Solution

Our abstraction-based system is generalizable to other categories of effects, but considers only operations on system resources, represented in a high-level language by operations through a foreign function interface (FFI). The FFI operations serve as base-level effects from which all other effects in the language are implemented. Thus, multiple low-level effects can be grouped under one high-level effect, which warrants fewer annotations, and allows effect representations to be tailored to different parts of the code, making their annotations more intuitive (but still valid) for reasoning about the program.

This effect system is being implemented in the Wyvern programming language [13]. Being a capability-safe language, Wyvern allows one to reason about effects "for free," trusting that changes in program state are limited to resources that the capability allows access to [10, 11]. The following is a sample program snippet in Wyvern:

```
1  resource type NetworkType
2    effect send = {} // no low-level effects
3    effect receive // to be defined by module
4    def sendData(data : String) : {send} Unit
5    def receiveData() : {receive} String
6
7  module def network() : NetworkType
8    effect send = {} // must be same as in type
9    effect receive = {java.ffiEffect}
10   def sendData(data : String) : {send} Unit
11     ...
12   def receiveData() : {receive} String
13     ...
14
15 module def dataProcessor(nw : NetworkType)
16   effect process = {nw.receive}
17   def processData() : {process} Unit
18     val result : String = nw.receiveData()
19     ...
20
21 // instantiate modules
22 val nw1 = network()
23 val nw1DataProcessor = dataProcessor(nw1)
```

Effects are declared with the keyword **`effect`**, and exist as part of a type signature or a module, similar to type members [4, 9, 14]. They are defined as either an empty set or a set of low-level effects (e.g., `java.ffiEffect`—a built-in, globally-available, base-level FFI effect), and a method header may include an effect annotation (such as an empty set) before its return type to report and enforce its effects. (There are

currently no expectations on the effects of a method without header annotation; section 4 elaborates on this.) In the example above, the `processData()` method of the `dataProcessor` module is annotated with the effect `process` on line 17, which is defined in the previous line to have the effect `receive` from the resource `nw` of type `NetworkType`. This is valid because `processData()` calls `nw.receiveData()`, which indeed has the effect `receive` defined by `NetworkType` on line 3.

Effects can be hidden from clients using type ascription, so that only effect definitions in the type, and not the module, are visible to higher-level code. In the example, because `network` modules have the type ascription "`:NetworkType`," an effect defined to have `nw.receive` will not be equivalent to one defined to have `java.ffiEffect` in `dataProcessor`. Effects in `nw` are hidden by the type ascription in this case.

To see the conceptual advantage of abstraction, compare `processData()` of `dataProcessor` with the following (from a version of the program without abstraction, e.g. declarations):

```
17  def processData() : {java.ffiEffect} Unit
```

The former is more intuitive, as the processing method has a "process" effect that accounts for the task and associated effects of "receiving" by a `NetworkType` resource, based on the context of this program. The use of `java.ffiEffect` in the latter appears jarring, and while one only needs to go back to `receiveData()` in the `network` module for its origin in this program, bigger programs would result in greater effort.

3 Implementation

I contributed to the system design, and am currently working on its implementation. Following the rest of the Wyvern compiler [1], which is written in Java, the current implementation parses effect declarations into ASTs, and translates the ASTs down to the intermediate language [13]. It then verifies the well-formedness of effects used in effect declarations and method headers, in both the type signatures and module definitions; if an effect cannot be found in the context at any of those points, an appropriate error is reported.

Finally, the compiler checks if a method has its header annotated (annotation set) with its actual effects (actual set). A method's actual set is the union of annotation sets of its method calls. To enable semantic checking and adhere to type ascription, both the actual set and annotation set are transformed to have the lowest level of effects in scope before comparison, by recursively looking up each effect's definition in the context. If the actual set is a subset of the annotation set, indicating that all of the method's real effects are included in its annotation (and allowing room for overcompensation), then the method's effect annotations are sound; otherwise, the compiler reports an appropriate error. To verify all annotated methods, the implementation still needs representation of base-level effects, such as `java.ffiEffect` in the code example, and ones defined by the programmer. The abstraction process, however, is currently functional.

[1] https://github.com/wyvernlang/wyvern

4 Future Work

In addition to base-level effects, future work would involve designing and implementing effect inference. Currently, methods without effect annotations in the headers are valid by default, as they make no claim regarding the effects to be expected, forcing the compiler to treat them as having no effects when they are called. Effect inference would enable effects to be verified for such methods, further reducing annotation overhead. Capabilities may be key to its design, as they help constrain the possible effects a method may have.

Once implementation is complete, the system may be evaluated by conducting user studies, such as examining how programmers would use Wyvern and the effect system to write programs requiring system resources. Observing them as they program can indicate how the system's mechanism interacts with their thought process, and the programmers' feedback can be used to inform its usability and utility for informal code analysis.

5 Related Work

Prior research has formulated different designs of effect systems. The Koka programming language implements polymorphic effects through row polymorphism and supports effect inference [6]. Devriese et al. have shown that effect parametricity can be used to support capability-based reasoning in JavaScript [2]. Bocchino et al. have developed a type and effect system to support parallel programming in Java, with region-path lists, index-parameterized arrays, subarrays, and invocation effects to guarantee determinism at compile time [1]. All these works offer a potentially advantageous design; however, they suffer from relatively large overhead, possibly due to requiring too many low-level effect annotations, which our approach aims at eliminating.

6 Conclusion

If the user studies result in positive responses to the design and implementation, then we will have identified a new, lightweight approach to facilitating program reasoning, through an abstraction-based effect system. Such approach could especially benefit the development process of large software systems, where overheads can be costly. Reasoning about system resource control would be more effective as well, aided by concise and tailored effect annotations.

Acknowledgments

This work was supervised by Darya Melicher and Dr. Jonathan Aldrich at Carnegie Mellon University (USA), and Dr. Alex Potanin at Victoria University of Wellington (New Zealand). It was supported by the NSF Research Experiences for Undergraduates grant.

References

[1] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. *SIGPLAN Not.* 44, 10 (Oct. 2009), 97–116. https://doi.org/10.1145/1639949.1640097

[2] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 147–162. https://doi.org/10.1109/EuroSP.2016.22

[3] David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

[4] Atsushi Igarashi and Benjamin C. Pierce. 1999. Foundations for Virtual Types. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*. Springer-Verlag, London, UK, UK, 161–185. http://dl.acm.org/citation.cfm?id=646156.679844

[5] Pierre Jouvelot and David K. Gifford. 1989. *Reasoning about continuations with control effects*. Vol. 24. ACM.

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

[7] John M. Lucassen. 1987. *Types and Effects Towards the Integration of Functional and Imperative Programming*. Technical Report. Massachusetts Inst of Tech Cambridge Lab for Computer Science.

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

[9] Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-oriented Programming. *SIGPLAN Not.* 24, 10 (Sept. 1989), 397–406. https://doi.org/10.1145/74878.74919

[10] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20

[11] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland.

[12] Flemming Nielson. 1996. Annotated Type and Effect Systems. *ACM Comput. Surv.* 28, 2 (June 1996), 344–345. https://doi.org/10.1145/234528.234745

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

[14] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A nominal theory of objects with dependent types. *ECOOP 2003–Object-Oriented Programming* (2003), 201–224.