Programming with Abstract Algebraic Effects

DARIUSZ BIERNACKI, University of Wrocław, Poland
MACIEJ PIRÓG, University of Wrocław, Poland
PIOTR POLESIUK, University of Wrocław, Poland
FILIP SIECZKOWSKI, University of Wrocław, Poland

We tackle the issue of abstraction in languages with algebraic effects and handlers, in the sense embodied by an ML-style module system. This issue turns out to be not as simple as hiding the definition of an effect from the user, since the execution of effectful programs rely on a dynamic process of matching an operation to the appropriate handler. This process, if implemented naively, can break the abstraction by "stealing" an effect from the inside of what was supposed to be a black box. In this presentation, we discuss the design principles and practice of programming in Helium: an experimental language that features abstract algebraic effects.

1 INTRODUCTION

Algebraic effects and handlers emerged from the theoretical work by Plotkin and Power [2004; 2001; 2002], and later by Plotkin and Pretnar [2013]. They now appear in a number of experimental languages, such as Eff [Bauer and Pretnar 2015], Frank [Lindley et al. 2017], and Koka [Leijen 2014], or as extensions of existing languages [Hillerström and Lindley 2016; Kammar et al. 2013].

One feature that – to our knowledge – is not yet present in any of these languages is an expressive module system that allows for abstraction over effects. Such an abstraction seems rather useful in practical programming, especially when one wants to achieve modularity in a larger-scale system. Consider a module `M` as a collection of functions, all sharing an effect, say a mutable cell. Often, we want this effect to be internal to these functions, which means that on the outside of the module it is revealed that the functions share an effect, but the user of the module cannot know what the effect is – hence, cannot interfere with it. This is particularly important, since – much like abstract types – it places restrictions on the *context* that uses the module: while concrete effects can be given *any* interpretation through a handler, the abstract effects can be handled only by using a handler provided by the module.

The technical difficulty in implementing abstract effects (which does not occur in the case of abstract types) arises from the dynamic nature of pairing an operation with a handler. Imagine that the user of the module `M` uses mutable state as an effect. Since the module's abstract effect is also instantiated to mutable state at runtime, the implementation needs to make sure that the user's handlers will not handle the module's state operations, and vice versa – that the module's handlers will not handle the user's operations.

To solve this problem, we propose to use *effect coercions*. Broadly speaking, the type checker knows which effect variable is existentially quantified, hence should be treated as fresh with respect to all other effects, such as the user's mutable state in the example above. If two effects potentially overlap (which is usually the case if at least one of them is abstract), the compiler places a coercion, which appropriately guides the process of matching operations to handlers at runtime, assuring that the abstraction is not broken. Our effect coercions generalise the *lift* operator in [Biernacki et al. 2018].

Authors' addresses: Dariusz Biernacki, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, dabi@cs.uni.wroc.pl; Maciej Piróg, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, mpirog@cs.uni.wroc.pl; Piotr Polesiuk, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, ppolesiuk@cs.uni.wroc.pl; Filip Sieczkowski, Institute of Computer Science, University of Wrocław, Joliot-Curie 15, Wrocław, 53-206, Poland, efes@cs.uni.wroc.pl.

In the talk, we would like to present some examples, to give a taste of programming with abstract algebraic effects, and discuss some of the design decisions and problems one faces when designing a programming language with effect abstraction.

2 THE LANGUAGE

To allow more experimentation with effect abstraction and its attendant coercion-based typing, we have implemented an experimental programming language, tentatively named Helium. The language supports advanced algebraic effects and handlers, sophisticated parametric polymorphism (including polymorphic records and constructors of algebraic data types), and type and effect abstraction through a simplified ML-style module system with signatures. The surface language does not include coercions, which are inferred during the type inference phase. After type inference, the program is translated to a language where subtyping constraints are explicit, similar in spirit to the recent work by Saleh et al. [2018], and down to a core calculus. From there, the types are erased, and programs can be run using an abstract-machine interpreter.

At the moment, the only impure part of the language is the I/O, which is tracked with a top-level abstract effect. Potential non-termination is not tracked (in contrast to, say, Koka), but a simple termination checker would be an easy extension – as nontermination is, semantically, an abstract effect anyway.

3 EXAMPLE: A SIGNATURE FOR UNION-FIND

Consider a standard ML signature for an imperative Union-Find data structure; see, for example, the `Util/uref` module in SML's standard library.[1]

```
type Set : type -> type

val new    : a -> Set a
val find   : Set a -> a
val union  : (a -> a -> a) -> Set a -> Set a -> Unit
```

A structure that matches this signature provides a conveniently abstracted interface to the union-find algorithm: we can create `new` abstract sets with a given representative, `find` a representative of a given set, and combine any two sets via the `union` operation – provided we know how to combine the representatives. The last operation is clearly effectful – after all, there is only one pure function into the unit type! – and in practice, so are the other. Thus, this seems like a prime case study for a language with algebraic effects. We stand to gain a lot: with an appropriate type system, the client could ensure that no side effects leak out. For instance, one can use this module to provide an implementation of Huet's unification algorithm [Huet 1976] that typechecks as pure, even though it uses union-find *internally*. First, however, we consider how we could adapt this signature to a language with algebraic effects.

One potential solution is to turn the *entire* signature into an algebraic effect. However, as we found out, this approach does not scale and we are not able to express the full power of the standard signature. Instead of following that approach, let us extend the signature with an abstract effect `UF`.[2]

```
type Set  : type -> type
effect UF : type -> effect

val new   : a ->[UF a] Set a
val find  : Set a ->[UF a] a
val union : (a -> a ->[r] a) -> Set a -> Set a ->[UF a, r] Unit
val handleUF : (Unit ->[UF a, r] b) ->[r] b
```

[1]We stray from the familiar notation, hopefully without loss of readability, as the present one generalises better to our system.
[2]Technically, `UF` is an effect *constructor*, parameterised by the representation type; for technical reasons this is necessary.

The arrow type now takes an additional annotation, specifying what effects a given function can perform – and the operations are, unsurprisingly, annotated with our abstract effect at the correct type. More importantly, in the specification of `union`, we see that the first argument itself can perform some unknown effects, and that the function itself performs both those, and additionally `UF`. Finally, the second new part of the specification is the presence of the `handleUF` – a *handler*, that can take any computation that may perform the abstract union-find effect, and turn it into a computation *without* this effect. Note that the handler is polymorphic both in the other effects the computation may perform, and the return type: its implementation simply records the allocated sets and performs their unions. Thus, we explicitly scope the part of the program that is allowed to use (an instance of) the union-find algorithm – a scoping that is necessary if we want to expose a pure interface of an (internally) impure computation.

REFERENCES

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: Relational interpretation of algebraic effects and handlers. *PACMPL* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, James Chapman and Wouter Swierstra (Eds.). ACM, 15–27. https://doi.org/10.1145/2976022.2976033

Gerard Huet. 1976. *Resolution d'equation dans les langages d'ordre 1, 2, . . . , ω.* Ph.D. Dissertation. Université de Paris VII, France.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.* 100–126. https://doi.org/10.4204/EPTCS.153.8

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 500–514. https://doi.org/10.1145/3009837

Gordon D. Plotkin and A. John Power. 2004. Computational Effects and Operations: An Overview. *Electronic Notes in Theoretical Computer Science* 73 (2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

Gordon D. Plotkin and John Power. 2001. Semantics for Algebraic Operations. *Electr. Notes Theor. Comput. Sci.* 45 (2001), 332–345. https://doi.org/10.1016/S1571-0661(04)80970-8

Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science)*, Mogens Nielsen and Uffe Engberg (Eds.), Vol. 2303. Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4:23 (2013), 1–36. https://doi.org/10.2168/LMCS-9(4:23)2013

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. 2018. Explicit Effect Subtyping. In *ESOP 2018 (to appear).*