Fram

Named Parameters Pushed to the Limit

PATRYCJA BALIK and PIOTR POLESIUK, University of Wrocław, Poland

We present Fram, an experimental programming language designed in the tradition of ML, and which aims to increase the ergonomics of programming with lexically-scoped handlers of algebraic effects. The key ingredient in the design of Fram is a very general mechanism of named parameters. It was originally inspired by the work of Lewis et al. on implicit parameters, but we observed that it can be generalized and, when integrated with other parts of the language, it is capable of expressing many sophisticated language features, such as records, existential types, or even functors. As a result, we obtained a useful, fully-fledged, yet relatively small programming language.

1 BACKGROUND

Algebraic effects with their handlers originally came from the world of denotational semantics [11, 12], but quickly gained an interest among designers of programming languages. Many experimental languages were developed (Koka [6], Links [5], Frank [8], Helium [1], and Effekt [3], just to name a few) and now algebraic effects are progressing towards industry-grade programming languages [15].

Algebraic effects and handlers initially had a dynamic semantics similar to exceptions, where each use of an effect is connected to the dynamically closest handler. However, this semantics is susceptible to problems such as accidental effect capture, and makes it difficult to use multiple instances of the same effect in a single piece of code. These problems can be solved by using lexical handlers [2, 16], where each handler introduces a fresh instance of a given effect. On the other hand, lexical handlers push the burden of connecting the use of an effect with its handler onto the programmer, which—without additional language features—results in code obscured by effect instance passing boilerplate.

Most implementations of lexical handlers [2, 4] conceal this inconvenience by some form of implicit parameter passing, where the implicit resolution is based on the type of the effect instance. We argue that this approach is unsatisfactory in the presence of multiple instances of the same effect, and instead propose an alternative solution inspired by name-based implicit parameters described by Lewis et al. [7]. We designed Fram[1] with the goal of exploring this idea, and in the process discovered that the mechanism can be generalized to subsume many additional language features.

2 NAMED PARAMETERS

During the talk we will focus on the design of our mechanism of named parameters and how we use it to express various language features. Similar to how polymorphism is dealt with by the Hindley-Milner type system, our language distinguishes types (assigned to expressions) and type schemes (assigned to variables). When a variable is used in an expression, its scheme is immediately instantiated to type. For instance, in Fram we can define polymorphic identity by writing `let id x = x`, which has the scheme `{type X} -> X -> X`, where `{type X}` binds the type variable `X`. Schemes can be used to bind other sorts of names, such as the following.

Named type parameters. We could alternatively assign the scheme `{X} -> X -> X` to our identity function. The difference is that now the type parameter has a specific name, which we can use for explicit type application, as in `id {X=Int} 42`.

Implicit parameters. Following the approach of Lewis et al., we incorporate information about implicit parameters into the scheme. For example, we could define a pretty-printing

[1]The current implementation of Fram is available at https://github.com/fram-lang/dbl.

function `pretty` that accepts its configuration as an implicit parameter. The scheme for `pretty` could then be something like `{'width : Int} -> Doc -> String`. The argument can be explicitly instantiated as `pretty {'width=80}`, or implicitly obtained from the environment based on its name.

Named parameters. Named parameters must always be explicitly provided, but their order does not matter. For example, a function `foo : {x : Int, y : Int} -> Int` can be called by writing `foo {y=13, x=42}`.

Optional parameters. Optional parameters are similar to named parameters, but their instantiation can be omitted, in which case a default value is taken. This is similar to optional parameters in OCaml, but since in Fram they are part of a scheme, rather than a type, we can avoid various awkward ambiguities. For example, we can define a function with the following signature.

```
parseList : {El, ?sep : String} -> Parser El -> Parser (List El)
```

In case we just return `parseList` as a value from a function, `?sep` will always be instantiated.

Our generalized approach to named parameters together with the clean separation of types and schemes integrates with other language features in an elegant way. For one, we can allow schemes to appear on the left-hand side of arrow types, giving us a form of Rank-N types [9, 10] Moreover, constructors of algebraic data types can have an associated scheme, rather than a type. This observation is particularly interesting, as it allows us to implement the following features with next to no effort.

Existential types. The first natural consequence, familiar from languages with GADTs, is that we can express existential types by giving constructors type parameters. For example, we can define the following type using the Church encoding of streams.

```
data Stream X = Stream of {St}, St, (St -> Pair X St)
```

In OCaml, there is no simple way to explicitly bind an existential type in a pattern matching. However, since `St` is a named type parameter, we can easily do this in Fram.

```
let head (Stream {St=T} (st : T) f) = fst (f st)
```

Records. In Fram, we can define record types using the following syntax.

```
data Vec = {x : Int, y : Int}
```

However, this is merely syntactic sugar for an algebraic data type with a single constructor with the fields as its named parameters (`data Vec = Vec of {x : Int, y : Int}`), along with automatic generation of selectors as regular methods.

Functors. If we assume a generative semantics of functors, then every use of a functor creates a new abstract type. We can model this by functions whose codomain is an existential type. For instance, the signature of finite sets can be expressed using the following existential data type.

```
data Set E = Set of
  { T, empty : T, add : T -> E -> T, mem : T -> E -> Bool, ... }
```

A functor can be represented using a polymorphic function that produces an element of type `Set E`.

```
make : { E, compare : E -> E -> Ord } -> Set E
```

So far, this approach is similar to how modules can be compiled out to System F_ω [13, 14]. In Fram, we allow named parameters to be instantiated in aggregate from a module. Similarly, they can be bound all at once under a given module name in a pattern matching. Therefore,

using this encoding directly gives us the power of a ML-style module system with only minimal overhead.

```
let Set { module IntSet } = make { module Int }
```

Interestingly, even if the module `Int` defines no type named `E`, the correct element type can still be inferred from the context.

3 CURRENT STATUS AND FUTURE PERSPECTIVES

The Fram project is under active development. As of the time of writing, the implementation of some of the described features is still in progress (optional parameters and encoding of functors). While we have initial results, they have not yet been merged into the repository. On top of the project's research origins, it is now also used for teaching purposes. Students actively participate in the language's implementation and discussions about its design. As a long term student project we are going to evaluate the language in practice—especially our approach to algebraic effects—by implementing a bootstrapped compiler.

REFERENCES

[1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *PACMPL* 3, POPL (2019), 6:1–6:28. https://doi.org/10.1145/3290319

[2] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *PACMPL* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

[3] Jonathan I. Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *PACMPL* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

[4] Jonathan I. Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *JFP* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

[5] 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

[6] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings of 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, April 12, 2014 (EPTCS, Vol. 153)*, Paul B. Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

[7] Jeffrey R. Lewis, Mark B. Shields, Erik Meijer, and John Launchbury. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19–21, 2000*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 108–118. https://doi.org/10.1145/325694.325708

[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/10.1145/3009837.3009897

[9] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21–24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 54–67. https://doi.org/10.1145/237721.237729

[10] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *JFP* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

[11] Gordon Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. *ENTCS* 73 (2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

[12] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *LMCS* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[13] Andreas Rossberg. 2015. 1ML — core and modules united (F-ing first-class modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1–3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 35–47. https://doi.org/10.1145/2784731.2784738

[14] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *JFP* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

[15] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. https://doi.org/10.1145/3453483.3454039

[16] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *PACMPL* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318