Submitted to MFPS 2024

The Functional Machine Calculus III: Choice

Early Announcement

Willem Heijltjes[1]

Department of Computer Science
University of Bath
Bath, United Kingdom

Abstract

The Functional Machine Calculus (Heijltjes 2022) is an extension of the lambda-calculus that preserves confluent reduction and typed termination, while enabling both call-by-name and call-by-value reduction behaviour and encoding the computational effects of mutable higher-order store, input/output, and probabilistic computation. This talk will present an extension of the FMC to capture exception handling, conditionals, constants, data types, and iteration.

Keywords: lambda-calculus, computational effects, exception handling

The study of computational effects in the λ-calculus is broad and varied. The overarching problem is that effects are *sequential* when modelled through global updates, while the λ-calculus is fundamentally *denotational*, governed by an equational theory. The contradiction manifests concretely as the loss of *confluence*, so that the choice of reduction strategy becomes salient, and syntactic control over evaluation behaviour becomes necessary. Semantic reasoning about programs, a main strength of the λ-calculus, is severely impaired—and much research has the objective of restoring it, including seminal contributions such as Moggi's monadic account of effects [8], Levy's *call–by–push–value* [7], Plotkin and Power's *algebraic effects* [9], and Plotkin and Pretnar's *effect handlers* [10]. Nevertheless, the problem is not settled; to quote Filinski in 2011:

> Yet few would confidently claim that programs with computational effects are by now as well understood, and as thoroughly supported by formal reasoning techniques, as types and terms in purely functional settings. — Filinski [3]

The Functional Machine Calculus (FMC) [4,1] is a new approach the problem of combining λ-calculus with effects. It takes a view of the λ-calculus as an instruction language for an abstract machine with a single stack in the style of Krivine [5], where *application* is *push*, *abstraction* is *pop*, and *variable* is *execute*. To accommodate effects, the calculus introduces the following two extensions [4].

Locations Multiple stacks on the machine, each named by a *location*, allow the encoding of various effects via push and pop actions: *mutable higher-order store*, as stacks of depth at most one; *input/output*, as pop-only respectively push-only streams; and *probabilities* and *non-determinism* as probabilistically respectively non-deterministically generated streams.

[1] Email: w.b.heijltjes@bath.ac.uk

Sequencing The introduction of *sequential composition* and its unit, imperative *skip*, gives control over evaluation behaviour away from strict call–by–name, and allows the encoding of Plotkin's call–by–value λ-calculus [11], Moggi's computational metalanguage [8], and call–by–push–value [7].

Encoding effects into the generalized operators of the calculus, rather than introducing primitives, means that two key properties of the λ-calculus are preserved.

Confluence Reduction in the FMC is confluent in the presence of effects. This is a consequence of the separation of operational behaviour, which governs the machine, from local reduction behaviour, which is the interaction of consecutive push and pop actions. Reduction equivalence for state then implements the algebraic laws of Plotkin and Power [9].

Types The FMC can be simply typed, which conveys strong normalization and termination of the machine. This gives a solution to the problem of typing higher-order store: *Landin's Knot* [6], which encodes recursion via higher-order store, cannot be typed (in its full generality).

In this talk I will introduce a third extension to the FMC, *choice*, which allows to include a wider range of computational behaviours: *constants*, *conditionals*, *data constructors*, *exception handling*, and *loops*. These have in common that, semantically, they are modelled by *sums* or *coproducts*: for example, the Booleans are given by the type $1 + 1$, the error monad is given by the functor $TX = E + X$ for a set of exceptions E, and loops are modelled by taking a map in $A \to A + B$ to one in $A \to B$ (looping on A, exiting on B) [2]. Together, these will be referred to as *choice* constructs.

The aim is sixfold. First and second, to preserve *confluence* and *types*: the resulting calculus should support a natural, confluent reduction relation, and a notion of simple types that guarantees termination of the machine and strong normalization of reduction (in the absence of loops). Third, *minimality*: choice constructs should be captured with as few syntactic operators as possible, avoiding any overlap in functionality and minimizing the interactions or reductions governing the semantics of the calculus. Fourth, *operational semantics*: the calculus should continue to be an instruction language for a simple and natural abstract machine. Fifth, *seamless integration*: different effects should combine seamlessly, without requiring lifting operations. Finally, the FMC has a natural *first-order restriction*, where function arguments are restricted to be (first-order) values, not arbitrary terms. The sixth aim is to preserve this restriction, which ensures that choice constructs are independent of the calculus being first-order or higher-order.

The approach has been to reconsider the notion of *choice* from first principles, with the aim of capturing coproducts and the constructs that they model in a simple and natural way, satisfying the six criteria above. This led to three (mostly) standard syntactic constructions, which however interact with the stack in subtle ways to give new and unexpected reduction behaviours. The resulting calculus is in some ways highly familiar, yet simultaneously in other ways novel and surprising.

In contrast with stateful effects, confluence and type safety are expected for exception handling. The main results are to integrate exceptions seamlessly with stateful effects, to support natural operational and denotational semantics, and to capture a wide range of behaviours with an elegant, minimal syntax.

References

[1] Barrett, C., W. Heijltjes and G. McCusker, *The Functional Machine Calculus II: Semantics*, in: B. Klin and E. Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023).
https://doi.org/10.4230/LIPIcs.CSL.2023.10

[2] Bloom, S. L. and Z. Ésik, *Iteration Theories - The Equational Logic of Iterative Processes*, EATCS Monographs on Theoretical Computer Science, Springer (1993), ISBN 978-3-642-78036-3.
https://doi.org/10.1007/978-3-642-78034-9

[3] Filinski, A., *Towards a comprehensive theory of monadic effects*, in: M. M. T. Chakravarty, Z. Hu and O. Danvy, editors, *Proc. 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, ACM (2011).
https://doi.org/10.1145/2034773.2034775

[4] Heijltjes, W., *The Functional Machine Calculus*, in: *Proceedings of the 38th Conference on the Mathematical Foundations*

of Programming Semantics, MFPS XXXXVIII, volume 1 of *ENTICS* (2022).
https://doi.org/10.46298/ENTICS.10513

[5] Krivine, J.-L., *A call-by-name lambda-calculus machine*, Higher-Order and Symbolic Computation **20**, pages 199–207 (2007).
https://doi.org/10.1007/s10990-007-9018-9

[6] Landin, P. J., *The mechanical evaluation of expressions*, The Computer Journal **6**, pages 308–320 (1964).

[7] Levy, P. B., *Call-by-push-value: A functional/imperative synthesis*, volume 2 of *Semantic Structures in Computation*, Springer Netherlands (2003).
https://doi.org/10.1007/978-94-007-0954-6

[8] Moggi, E., *Notions of computation and monads*, Information and Computation **93**, pages 55–92 (1991).

[9] Plotkin, G. and J. Power, *Notions of computation determine monads*, in: *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 342–356, Springer, Berlin, Heidelberg (2002).

[10] Plotkin, G. and M. Pretnar, *Handling algebraic effects*, Logical Methods in Computer Science (2013).

[11] Plotkin, G. D., *Call-by-name, call-by-value and the λ-calculus*, Theoretical Computer Science **1**, pages 125–159 (1975).