# A New Functional Language with Effect Types and Substructural Handlers

Pavel Sokolov
*Faculty of Computer Science*
*Higher School of Economics*
Moscow, Russia
ppsokolov@edu.hse.ru

***Abstract*—We propose a new programming language where algebraic effects are visible to the type system as polymorphic effect types. A substructural type system guarantees a predictable behavior of effectful computations; we model these using row polymorphism and recursive types. The result of the transformation lets us represent effectful programs in the continuation-passing style, which should be familiar to those who are acquainted with Freer monads. As a proof of concept, we will implement an interpreter in Haskell, which we consider to be one of the main inspirations for this project, with a self-hosting compiler on the way.**

***Keywords*—algebraic effects, substructural type system, row polymorphism, equirecursive types**

## I. Introduction

This project belongs to the area of programming language design. We aim to develop a new functional language featuring a modern type system for ergonomic management of side effects and program resources.

Simply put, side effects are everything a program does to the environment: device management, reads from / writes to files, missile launches, etc. Programming errors in this area range from embarrassing bugs to outright catastrophes; every substantial development team wants to be sure that these errors never happen. However, existing mainstream programming techniques such as testing and code review are unsound, which means that you can never be sure that there are no bugs even if all tests have passed.

Luckily, there is a sound method known to every seasoned programmer: type system expressive enough to statically represent invariants you wish to uphold. In this project, we implement a new language with effect types representing computations that influence its environment. The idea is not novel: there are research languages Unison [1] and Koka [2] on this topic; a simpler and somewhat more tractable model is studied in [3]. Our innovation is in the usage of a substructural type system to statically verify that continuations in effect handlers are called an appropriate number of times. A concept for such combination is folklore and mentioned in [4]; this project properly expands on it.

The expected results are twofold. Firstly, we will present a profound design of a sound substructural type system for a functional language with algebraic effects. Secondly, we will provide a proof-of-concept interpreter in Haskell. Formal verification of soundness and implementation of efficient self-hosting compiler are reserved for future work.

We begin with a general analysis of effect systems including the ones mentioned above as well as their analogues in Haskell. Then we overview the features in our language that aid algebraic effects. Before concluding, we provide a basic description of the type system and core typing rules.

## II. Literature Review

The point of pure functional programming is to give a programmer precise control over side effects that happen during program execution. The consensus is to use special kinds of monads, notable examples being monad transformers [5], free monads [6] and, most recently, freer monads. The last ones in conjunction with heterogeneous lists give rise to the Extensible Effects approach [7] which, as claimed by the authors, overcomes the drawbacks of previous tools. However, these techniques are rarely used outside of Haskell because they require quite advanced type-level machinery which is simply not available in other, either more simple or mainstream, languages.

Albeit surprisingly, the key concept behind Extensible Effects is simple. It is to view an effect as an interaction between a sub-expression and a central authority that handles effect execution. Languages with algebraic effects, for their part, capitalize on this idea and provide built-in language primitives to build both such sub-expressions — from now on, we will call them "procedures" — and central authorities, "handlers". Communication between them is done via continuation-passing [8].

For example, Multicore OCaml has a way to declare new effects as well as a builtin function `perform` to, indeed, *perform* an effect [9]. Handler is just a regular `case` statement that pattern-matches on performed effects. However, for a long time effects were invisible to OCaml's type system; this means that every function could be a procedure executing arbitrary effects, which is in no way different from the mainstream approach.

As for Unison, they introduce proper effect types in the form of attributes on the function type. However, they do not restrict how many times a continuation is called in a handler [10], which may lead to bizarre and unexpected behavior of a program, just like a program in C which uses `fork`.

Koka's treatment of continuations is only slightly better as they do not let a programmer access continuation directly, which enables them (and makes them) to introduce a whole lot of ad-hoc mechanisms to make both behavior and performance of programs more predictable [11].

A more disciplined solution is offered in [3]. The authors of Runners in Action simplify the framework by forcing continuations to be in tail-call positions, effectively replacing continuation with a simple `return` statement (pun intended). However, it makes certain effects inexpressible, most notably nondeterminism and asynchrony.

## III. Methodology

### A. General description

We get rid of the tradeoffs mentioned above by embracing a substructural type system. In its setting, every value can have two constraints imposed on its usage: can it be silently dropped? Can it be copied? We simply make a programmer specify the constraints for continuations while declaring new effects. As a bonus, these constraints can be later used by an optimizing compiler to place values outside of the garbage-collected heap.

Actually, there already are a few different substructural type systems. Linear Types extension in Haskell [12] and unique types in Rust [13] are the most well-known examples, although we take a slightly different approach which is extensively described in [14]. As of our current knowledge, this project is the first to utilize URAL for language with algebraic effects.

Let us now pay attention to the ergonomics of the language. Effect types themselves make type signatures more complicated than those in mainstream languages; to make matters worse, linear types are notorious for their brimming function signatures. This calls for global type inference. To separate concerns about type inference and syntactic sugar from the rest of the language, we followed in the footsteps of Glasgow Haskell Compiler and split the language in two parts [15]: surface language for developers to write programs in and core language for type checking and evaluation algorithms to run in.

In order to encode inductive data types of a surface language, we, perhaps mistakingly, decided to use equirecursive types as they are defined in [16]. However, we discovered that this opens a possibility to eliminate algebraic effects from the core language altogether: coupled with row polymorphism like in [17] and [18], equirecursive types allow us to desugar types of computations into open unions where each variant represents one effectful operation; this encoding is very similar to Freer monads [7]. After employing this encoding, effect handlers become nothing more than recursive functions which accept computations and pattern-match on them. Nice!

One should note, though, that this encoding is not the only reason why we incorporated row types. It simplifies core language: without it, one would typically need to introduce a binary operation on types and unit of it for every kind of composite data structure (multiplicative conjunction, additive conjunction, additive disjunction), which, together with the introduction and elimination rules, would comprise a whole lot of logical rules. With our generalized flavor of row types, we needed only three rules for each kind; the idea of generalization itself stems from the similarities between records for both multiplicative and additive conjunctions.

$$\frac{k \text{ kind}}{K,\ a :: k \ \vdash\ a :: k}\textsc{KVar}$$

$$\frac{k, k' \text{ kind} \qquad K \ \vdash\ x :: k \qquad K \ \vdash\ t \ ::\ k \to k'}{K \ \vdash\ t\ x :: k'}\textsc{KApp}$$

$$\frac{k, k' \text{ kind} \qquad K,\ a :: k \ \vdash\ b :: k'}{K \ \vdash\ (a :: k.\ b) :: k \to k'}\textsc{KAbs}$$

$$\frac{K \ \vdash\ a :: k \qquad K \ \vdash\ b :: k'}{K \ \vdash\ \langle a, b\rangle :: k \times k'}\textsc{KPair}$$

$$\frac{K \ \vdash\ p \ ::\ k \times k'}{K \ \vdash\ \text{fst}\ p :: k}\textsc{KFst} \qquad \frac{K \ \vdash\ p \ ::\ k \times k'}{K \ \vdash\ \text{snd}\ p :: k'}\textsc{KSnd}$$

$$\frac{k \text{ simple kind} \qquad K,\ a :: k \ \vdash\ b :: k}{K \ \vdash\ (\mu\ a.\ b) :: k}\textsc{KFix}$$

$$\frac{k, k' \text{ kind} \qquad K \ \vdash\ r :: \text{row}\ k \qquad K \ \vdash\ t \ ::\ k \to k'}{K \ \vdash\ t\ @\ r \ ::\ \text{row}\ k'}\textsc{KMap}$$

Fig. 1. Type-level operations

### B. Typing rules

Having reached a general understanding of our type system, let us describe typing rules of a core language.

*1) Kinds:* Our core language has the following kinds:

$$\begin{aligned} L &::= \ \text{pretype} \mid \text{type} \mid L \times L \\ K &::= \ L \mid \text{mult} \mid \text{row}\ K \mid K \times K \mid K \to K \end{aligned}$$

Kinds in $L$ are called "simple kinds", whereas kinds in $K$ are "proper kinds" or just "kinds".

Types are pretypes annotated with multiplicity modality (terms of kind mult). $\times$ is a kind constructor for type-level pairs, which is extremely useful for both mutually recursive types and desugaring of effect types. Note that row types have another kind as a parameter: they are used to represent a labeled collection of any kind, not only of types.

*2) Type-level operators:* Rules for type-level operations are presented in Fig.1. Kinding context is supposed to be a usual context supporting exchange, weakening, and contraction. Once we introduce type-level operations, we also have to determine operational semantics (which are fairly obvious) and sound equivalence relations. This is the whole point of [16]; we only have to extend algorithms to handle polymorphic multiplicities and rows, which are elaborated further. General type-level expressions are later denoted as $E$.

*3) Multiplicities:* Multiplicities form a distributive lattice on four elements:

$$M ::= \quad E \mid ! \mid ? \mid \ + \ \mid \ * \ \mid M \vee M \mid M \wedge M$$

Multiplicity literals draw an analogy with the notation of regular expressions. $\vee$ is used to join constraints in composite structures; $\wedge$ is used to desugar constraints on multiplicity variables. For example, a type of function duplicating its argument is desugared from the surface language like this:

$$\begin{aligned}[m \leq +] \to a^m \to (a^m, a^m) \quad &\rightsquigarrow \\ &\big(\text{Forall}(a :: \text{pretype})(m :: \text{mult}). \\ &\quad (a^{m \wedge +} \to \{.0 : a^{m \wedge +}, .1 : a^{m \wedge +}\})^*\big)^* \end{aligned} \tag{1}$$

General expressions for multiplicities are strongly normalizing, therefore we can evaluate them eagerly. The subsumption of multiplicities can be decided in quadratic time by a straightforward recursive algorithm. Equality of multiplicities can be reduced to checks of two subsumptions.

*4) Row types:* Rows form a join semilattice:

$$R ::= \quad E \mid \varnothing \mid (.n_j : t_j, ...R) \mid R \vee R$$

One should note that, because of the substructural type system, we cannot silently drop repeating entries as is done in JavaScript. Therefore rows can have duplicated labels; however, we provide no disambiguation between duplicated entries because this would not respect the absence of order between them.

Checking the equality of two rows is not different from deciding the equality of two multimaps.

*5) (Pre)types:* Pretypes contain functions, forall binders and composite types ($R$ below are assumed to be rows of types):

$$\begin{aligned} T ::= &\quad E \mid P^M \\ P ::= &\quad E \mid T \to T \mid \Pi(x :: K).T \\ \mid &\quad \{...R\} \mid [...R] \mid (|...R|) \end{aligned}$$

We do not expect any complications while adapting the algorithm for checking type equality from [16].

*6) Modelling effect types in the core language:* Now, we are ready to define the algebra of effect types as well as desugaring for their expressions. The algebra looks very similar to the algebra of row types:

$$\text{Eff} ::= \quad E \mid \varnothing \mid (op : T \xrightarrow{M} T, ...\text{Eff}) \mid \text{Eff} \vee \text{Eff}$$

In addition, just like with row types, effect types can contain duplicate entries. In [17] it is argued that, actually, this is a good thing.

Let us denote the type of computation producing effects in $e$ and returning value of type $a$ as $e \models a$. The desugaring procedure goes as follows:

$$\begin{aligned} e &\rightsquigarrow e' \\ f\ t &:= \langle u, m, v\rangle \Rightarrow (u, v \xrightarrow{m} t)^{\text{mult } u \vee m} \\ e \models a &\rightsquigarrow \mu t.(|.\text{pure} : a, ...(f\ t \ @\ e')|) \end{aligned} \tag{2}$$

As expected from the similarity with rows, effect types are first desugared into rows of triples: type $\times$ mult $\times$ type.

$$\frac{K;\ \Gamma,\ x : t,\ y : u,\ \Delta\ \vdash\ e : T}{K;\ \Gamma,\ y : u,\ x : t,\ \Delta\ \vdash\ e : T}\ \Gamma\text{-Exchange}$$

$$\frac{m \leq ? \qquad K;\ \Gamma\ \vdash\ e : T}{K;\ x : p^m,\ \Gamma\ \vdash\ e : T}\ \Gamma\text{-Weakening}$$

$$\frac{m \leq + \qquad K;\ x : t^m,\ x : t^m,\ \Gamma\ \vdash\ e : T}{K;\ x : t^m,\ \Gamma\ \vdash\ e : T}\ \Gamma\text{-Contract}$$

$$\frac{t\ \text{type}}{K;\ x : t\ \vdash\ x : t}\ \text{TVar}$$

$$\frac{K;\ \Gamma,\ x : t\ \vdash\ e : u \qquad K\ \vdash\ \sup \Gamma \leq m}{K; \Gamma\ \vdash\ (\lambda x.\ e) : (t \to u)^m}\ \text{TAbs}$$

$$\frac{K; \Gamma\ \vdash\ f : (t \to u)^m \qquad K; \Delta\ \vdash\ x : u}{K;\ \Gamma, \Delta\ \vdash\ f\ x : u}\ \text{TApp}$$

$$\frac{a :: k,\ K;\ \Gamma\ \vdash\ e : t \qquad K\ \vdash\ \sup \Gamma \leq m}{K; \Gamma\ \vdash\ (\Lambda a :: k.\ e) : (\Pi(a :: k).t)^m}\ \text{TGen}$$

$$\frac{K; \Gamma\ \vdash\ f : (\Pi(a :: k).t)^m \qquad K\ \vdash\ d :: k}{K; \Gamma\ \vdash\ f\ d : \{a/d\}t}\ \text{TInst}$$

$$\frac{\begin{array}{c}\forall n \in \text{keys } R : \quad K; \Gamma_n\ \vdash\ e_n : R[n] \\ K\ \vdash\ \sup R \leq m\end{array}}{K; \Gamma_n\ \vdash\ \{.n : e_n\} : \{\dots R\}^m}\ \text{TAndI}$$

$$\frac{K; \Gamma\ \vdash\ s : \{\dots R\}^m \qquad K;\ \Delta, R\ \vdash\ e : t}{K;\ \Gamma, \Delta\ \vdash\ \textbf{let}\ \{.1, \dots, .n\} = s\ \textbf{in}\ e : t}\ \text{TAndE}$$

$$\frac{\begin{array}{c}\forall n \in \text{keys } R : \quad K; \Gamma\ \vdash\ e_n : R[n] \\ K\ \vdash\ \sup R \leq m\end{array}}{K; \Gamma\ \vdash\ [.n : e_n] : [\dots R]^m}\ \text{TWithI}$$

$$\frac{K; \Gamma\ \vdash\ w : [\dots R]^m \qquad n \in \text{keys } R}{K; \Gamma\ \vdash\ w.n : R[n]}\ \text{TWithE}$$

$$\frac{K; \Gamma\ \vdash\ e : t^{m'} \qquad K\ \vdash\ m' \leq m \qquad n \in \text{keys } R}{K; \Gamma\ \vdash\ .n\ e : (|\dots R|)^m}\ \text{TOrI}$$

$$\frac{\begin{array}{c}K; \Gamma\ \vdash\ o : (|\dots R|)^m \\ \forall n \in \text{keys } R : \quad K;\ x : R[n],\ \Delta\ \vdash\ e_n : t\end{array}}{K;\ \Gamma, \Delta\ \vdash\ \textbf{case}\ o\ \textbf{of}\ (.nx := e_n) : t}\ \text{TOrE}$$

Fig. 2. Context and Terms

*7) Terms:* Rules for terms follow straightforwardly from rules for (pre)types and are provided in Fig.2. Rules for substructural term context are provided in the same figure. Note that while defining operations on rows, we have to account for polymorphic rows, though it is quite simple.

## IV. Results

It is expected to will have implemented an interpreter of the language described above, including the split in the surface and core languages: logical rules of the core language are already properly written out, and the Haskell ecosystem has all the necessary tools.

Type inference and the existence of principal types in the surface language are highly anticipated because both row types and substructural types play nicely with Hindley-Milner. However, general equirecursive types may pose a threat to the type inference, therefore they might be hidden beneath the surface.

## V. Conclusion

Utilising the recent advancements in type theory, we developed a polished model of computations with side effects which is an improvement upon analogues in terms of soundness and, theoretically, performance. While doing so, we made initially unrelated constructs from different type systems interplay nicely: row polymorphism with substructural typing, both type-level pairs and row polymorphism with algebraic effects... In addition, we saved ourselves the headache of developing a dedicated runtime for algebraic effects using a well-typed encoding into continuation-passing style.

The most notable limitation of our solution is that core language is hardly extensible: generalized algebraic data types cannot be easily fit into it; coexistence of dependent types with substructural types is still an ongoing research topic;... To overcome this limitation, we can make use of the two-layered architecture of the language and change core language independently from the surface to incorporate new features.

Nevertheless, as it was said in the beginning, the primary goals after this project are to formally verify the soundness of the type system and to write a self-hosting compiler. The first one is needed primarily to exclude the human factor; writing a compiler is required to fulfill promises of performance.

Designing one more programming language might not change anything in the grand scheme of things; however, in order to reach the ideal future where every programmer writes programs in languages with effect types, we as a whole programming community need enough failed attempts to learn from and working alternatives to choose from.

## References


[1] Unison Computing, "The Unison language," 2021, last accessed 24 February 2022. [Online]. Available: https://www.unisonweb.org/

[2] D. Leijen, "Algebraic effects for functional programming," Microsoft Research, Tech. Rep. MSR-TR-2016-29, August 2016.

[3] D. Ahman and A. Bauer, "Runners in Action," *Lecture Notes in Computer Science*, p. 29–55, 2020.

[4] K. Sivaramakrishnan, "Effective concurrency with algebraic effects," 2015, last accessed 24 February 2022. [Online]. Available: https://kcsrk.info/ocaml/multicore/2015/05/20/effects-multicore/

[5] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, 1995, p. 97–136.

[6] E. Kmett, "Monads for free," 2008, last accessed 24 February 2022. [Online]. Available: http://comonad.com/reader/2008/monads-for-free/

[7] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," *SIGPLAN Not.*, vol. 50, no. 12, p. 94–105, aug 2015.

[8] M. Pretnar, "An introduction to algebraic effects and handlers. Invited tutorial paper," *Electronic Notes in Theoretical Computer Science*, vol. 319, pp. 19–35, 2015, the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[9] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy, "Retrofitting effect handlers onto OCaml," *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Jun 2021.

[10] Unison Computing, "Unison language reference. Pattern matching on ability constructors," 2021, last accessed 24 February 2022. [Online]. Available: https://www.unisonweb.org/docs/language-reference#pattern-matching-on-ability-constructors

[11] D. Leijen, "The Koka programming language. Effect handlers," 2022, last accessed 24 February 2022. [Online]. Available: https://koka-lang.github.io/koka/doc/book.html#sec-handlers

[12] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear Haskell: practical linearity in a higher-order polymorphic language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 1–29, Jan 2018.

[13] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2017.

[14] A. Ahmed, M. Fluet, and G. Morrisett, "A step-indexed model of substructural state," *SIGPLAN Not.*, vol. 40, no. 9, p. 78–91, sep 2005.

[15] S. Marlow and S. Peyton-Jones, "The Glasgow Haskell Compiler," in *The Architecture of Open Source Applications, Volume II*, A. Brown and G. Wilson, Eds. CreativeCommons, 2012, pp. 67–88.

[16] C. A. Stone and A. P. Schoonmaker, "Equational theories with recursive types," Harvey Mudd College, Tech. Rep., 2005.

[17] D. Leijen, "Koka: Programming with row polymorphic effect types," *Electronic Proceedings in Theoretical Computer Science*, vol. 153, p. 100–126, Jun 2014.

[18] J. Garrigue, "Programming with polymorphic variants," in *ACM SIGPLAN Workshop on ML*, Baltimore, MD, 10 1998.