Contents

Abstract

We propose a new programming language where algebraic effects are visible to the substructural type system as polymorphic effect types. This report includes a high-level specification of a surface language along with description of procedures that translate it into OdLang Core, a small explicitly typed interpreted language.

Мы предлагаем новый язык программирования с субструктурной системой типов, в которой алгебраические эффекты представлены полиморфными типами эффектов. Данный отчёт содержит высокоуровневую спецификацию языка вместе с описанием процедур трансляции этого языка в OdLang Core, маленький явно типизированный интерпретируемый язык.

Keywords: algebraic effects, substructural type system, bidirectional type checking.

Chapter 1

Introduction

This is part of a project in 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 the topic.

In the previous line of work, we decided to split the language in two layers: a surface language, OdLang, to write programs in and a core language, OdLang Core, to check types and interpret expressions. We also described logical

rules for the core language; this report includes a short recap of Core's most important features. In the main part of this paper we come up with syntactic rules of OdLang as well as with algorithms for translating it into OdLang Core, including multiplicity elision rules, implicit quantification, bidirectional type checking, desugaring of multiplicity constraints and effect types and, finally, desugaring of terms.

Chapter 2

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.

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.

Chapter 3

OdLang Core

Before diving into syntactic richness of a surface language, let us recap the most important bits about core language.

3.1 Kinds

OdLang Core is a dialect of System F_ω. It has six kinds: substructural multiplicities, rows (type-level dictionaries) of any kind, pretypes (algebraic data types, function types), types (pretypes with a specified multiplicity), type-level pairs and operators (type-level functions between kinds). (Pre)types can be equirecursive, for this we adapt an algorithm described in [15].

In addition to the expected operations, we introduce a way to apply type operator of kind $k \to k'$ to the row of kind "row k", yielding an expression of kind "row k'". This way "row" can be perceived as a kind-level functor.

Precise kinding rules for type-level calculus are given in 3.1.

3.2 Multiplicities

Multiplicities form a distributive lattice on four elements:

- **U**nrestricted ($*$) — values of such types can be left unused and can be arbitrarily copied;
- **R**elevant ($+$) — cannot be left unused;

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

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

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

$$\frac{K \ \vdash\ a :: k \qquad K \ \vdash\ b :: k'}{K \ \vdash\ \langle a, b\rangle :: k \times k'}\text{KPair} \qquad \frac{K \ \vdash\ p \ ::\ k \times k'}{K \ \vdash\ \text{fst}\ p :: k}\text{KFst}$$

$$\frac{K \ \vdash\ p \ ::\ k \times k'}{K \ \vdash\ \text{snd}\ p :: k'}\text{KSnd} \qquad \frac{k \text{ simple kind} \qquad K,\ a :: k \ \vdash\ b :: k}{K \ \vdash\ (\mu\ a.\ b) :: k}\text{KFix}$$

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

Figure 3.1: Type-level operations

- **A**ffine (?) — cannot be copied, but can be left unused;
- **L**inear (!) — must be used exactly once.

Multiplicity literals draw an analogy with the notation of regular expressions. We can also join constraints of two multiplicities using $\vee$ and take common constraints using $\wedge$. Multiplicity polymorphism is present, too.

3.3 Row types

As it has been mentioned before, rows are type-level dictionaries. To refine on this statement: rows of kind "row k" are *multi*-maps from field names to type-level expressions of kind k. We can declare row literals as well as row variables (that is, we have row polymorphism) and join rows together. However, there is no polymorphism over field names.

3.4 Pretypes

Pretypes include simple functions as well as `forall` binders for different kinds of polymorphism. In addition, any row of types R can be turned into an algebraic data type:

- $\{\ldots R\}$ is a record of values in R. To construct a value of record type is to provide all its fields, to use the record is to use all its fields.
- $[\ldots R]$ is a list of alternatives listed in R. To construct it is to provide all its fields built from the same context, to use it is to choose one of alternatives.
- $(|\ldots R|)$ is a tagged union of options listed in R. To construct it is to construct one of the options, to use it is to pattern-match on it and describe how to use every option.

Precise typing rules are given in Fig.3.2-3.3.

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

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

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

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

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

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

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

Figure 3.2: System-F Fragment and Substructural Context

$$\frac{\forall n \in \text{keys } R : \quad K; \Gamma_n \ \vdash \ e_n : R[n] \qquad K \ \vdash \ \sup R \leq m}{K; \overline{\Gamma_n} \ \vdash \ \{\overline{.n : e_n}\} : \{\ldots R\}^m} \textsc{TAndI}$$

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

$$\frac{\forall n \in \text{keys } R : \quad K; \Gamma \ \vdash \ e_n : R[n] \qquad K \ \vdash \ \sup R \leq m}{K; \Gamma \ \vdash \ [\overline{.n : e_n}] : [\ldots R]^m} \textsc{TWithI}$$

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

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

$$\frac{K; \Gamma \ \vdash \ o : (|\ldots R|)^m \qquad \forall n \in \text{keys } R : \quad K; \ x : R[n], \ \Delta \ \vdash \ e_n : t}{K; \ \Gamma, \Delta \ \vdash \ \textbf{case} \ o \ \textbf{of} \ (\overline{.n \ x := e_n}) : t} \textsc{TOrE}$$

Figure 3.3: Terms of Algebraic Data Types

Chapter 4

Surface Language

OdLang itself is a syntactic extension over its core incorporating a new kind (kind of effects), a new type constructor (effectful computation) and a few features revolving around recursion and pattern-matching. Last but not least, while core language is typed explicitly, surface language has a few tweaks to reduce amount of type annotations and improve their readability.

4.1 Type-level quality of life improvements

Here we list more or less expected and conventional features of a modern functional programming language along with novel pieces of syntactic sugar.

1. Core has only type-level pairs. However, OdLang has arbitrary nonempty and nonsingular type-level tuples:

   ```
   tuple Quadruple : type * multiplicity * pretype * type :=
       < {}*, !, String, String? >
   ```

   Desugaring is trivial: `*` is a right-associative `x`; components of tuples are folded with `<_,_>` from right to left.

   To access components of a type-level tuple, one can pattern-match on it:

   ```
   tuple < _, _, Third, _ >
       : type * multiplicity * pretype * type
       := Quadruple
   ```

Desugaring is also trivial and involves repeated applications of `snd` and one application of `fst` for each component except for the last.

2. Using type-level `fix` explicitly is cumbersome. One can build a dependency graph of type-level expressions and use `fix` for strongly connected components. The same approach is used on the term level for recursive functions and was originally described in [16].

   Before desugaring:

   ```
   type Fix (f : type -> type) := f (Fix f)

   type Red := { head : Int*, tail : (() -> Blue)* }*
   type Blue := { head : Int*, tail : (() -> Red)* }*
   ```

   After desugaring:

   ```
   type Fix (f : type -> type) := (fix t : type . f t)

   tuple <Red, Blue> : type * type :=
       (fix <r, b> : type * type . <
           { head : Int*, tail : (() -> b)* }*,
           { head : Int*, tail : (() -> r)* }*,
       >)
   ```

3. In most cases, kind of operator and forall arguments can be inferred from its usage in the resulting expression:

   ```
   pretype Beside f r := {...r, ...(f @ r)}
       -- r is inferred to be a row of types
       -- f is inferred to be a (type -> type) operator
   ```

   The case where inference fails can be seen above, in `Fix` case, because `f` can be kinded as `pretype -> pretype`, too.

   To perform said kind inference, one can utilize a standard unification algorithm [17].

4. Following convention in Haskell, we can forbid lowercase type-level entities and assume that every lowercase name in a type is an argument of a forall binder.

5. Omnipresent multiplicity annotations obscure type signatures. However, certain annotations can be omit. For example, annotations on a record and on nearly all arrows are elided here:

```
type Fmap f m := forall a b. (a -> b) % m -> f a -> f b


pair : a -> b -> { fst: a, snd: b } :=
    | x, y := { fst: x, snd: y }
```

   Following typing rules, we can only generate constraints on annotations, but not annotations themselves. Therefore we cannot infer them, but we can introduce sane defaults for common cases, as is done with lifetime annotations in Rust [18]. We propose two elision rules:

   (a) For arrow and forall types, choose the most permissive annotation;

   (b) For algebraic data types, introduce a new annotation variable.

   Applying elision rules to the expressions above, we get:

```
type Fmap f m := (forall a. (forall b.
    ((a -> b) % m -> (f a -> f b) % m)*
)*)*


pair : (forall a. (forall b . (forall m .
    (a -> (
        b -> { fst: a, snd: b } % (m \/ mult a \/ mult b)
    ) % mult a)*
)*)*)* :=
    ...
```

6. OdLang Core allows records, but not tuples. Tuples in surface language are encoded using a record type of sufficient arity:

```
type Tuple4 a b c d := { n1: a, n2: b, n3: c, n4: d }
```

Instead of doing this, we could represent tuples as nested pairs, but this representation does not bring any immediate benefits.

7. OdLang Core does not have a way to constrain multiplicities. However, we can use `/\` to relax constraints multiplicities impose themselves:

```
dup : (a <= +) => a -> (a, a) :=
    | x -> (x, x)

dup' : a % (m /\ +) -> (a % (m /\ +), a % (m /\ +)) :=
    | x -> (x, x)
```

4.2 Term-level quality of life improvements

Of course, OdLang includes nested pattern-matching and pattern-matching of product types. In addition, we include `let` expressions and `where` blocks as they are in Haskell.

Pattern-matching has to be exhaustive. Recursive functions can be defined via Y combinator (or any other fixpoint combinator) which is typeable in the Core language thanks to equirecursive types.

1. Named functions admit a more compact declaration:

```
row Fin2 a b where
    fst: a
    snd: b

pretype Pair a b := { ...(Fin2 a b) }

pair : a -> b -> Pair a b :=
    | x, y := { fst: x, snd: y }

pair' (x : a) (y : b) : Pair a b :=
```

```
    { fst: x, snd: y }
```

2. Lambda expressions followed by a `case` statement admit a more compact declaration:

```
type List m a := (|
    Nil: {} % m
    Cons: (a, List m a) % (m \/ mult a)
|) % (m \/ mult a)

foldr (f : a -> b -> b) (s : b) : List m a -> b :=
    | xs := case xs of
        Nil _ := s
        Cons (x, xs) := f x (foldr f s xs)

foldr' (f : a -> b -> b) (s : b) : List m a -> b :=
    | Nil _ := s
    | Cons (x, xs) := f x (foldr' f s xs)
```

3. Dot notation is available both for `[]`-types and for records:

```
ifThenElse : Bool -> [...(Fin2 a a)] -> a :=
    | True, opts := opts.fst
    | False, opts := opts.snd
    -- already in Core, desugaring is not needed

join (f : a -> b -> c) (p : Pair a b) : c := f p.fst p.snd

join' : (a -> b -> c) -> Pair a b -> c :=
    | f, p := let { fst, snd } := p in f fst snd
```

4.3 Algebraic effects

4.3.1 Effect types

As you might remember, the biggest hurdle of this project is trying to incorporate algebraic effects into a sound substructural type system. We start with introducing a new kind, `effect`:

```
effect State s where
    modifyState: (s -> s)! ->! {}*

effect Generator t where
    yield: t ->! {}*
```

Effect is a list of operations a corresponding computation can do. From imperative point of view, each operation is a side-effectful function. Multiplicity annotation after its arrow describes an amount of times a control flow might return to the point where operation was called.

Effects can be joined: `State (List ! t) & Generator t`.

Denoting the type of a computation requiring effect `e` and yielding a result of type `a` as `e |= a`, let us describe the desugaring of effects into Core language.

Effects themselves are rows of triples `type * multiplicity * type`:

```
row State s where
    modifyState: < (s -> s)!, !, {}* >

row Generator t where
    yield: < t, !, {}* >
```

Therefore joining effects is the same as joining rows. In addition, effect polymorphism becomes a special case of row polymorphism.

Given an effect row `e` and a type `a`, `e |= a` is best described as an operator:

```
type intoEntry rec <from, m, to> :=
    (from, (to -> rec % (m \/ mult to)) % m) % (m \/ mult from)
```

```
pretype (e |= a) := (|
    pure: a,
    ...(intoEntry (e |= a) @ e)
|)
```

4.3.2 Computations

While on term level, effectful operations are typed as functions accepting its argument and continuation. For example, `yield` is typed as

```
t -> ({}* -> (Generator t & e |= a)!)! -> Generator t & e |= a
```

One can see that this is minor syntactic sugar over creation of a corresponding variant of a union. However, passing continuations as is leads to well-known issue of callback hell:

```
countToThree : Generator Nat |= Unit :=
    yield 1 | _ :=
        yield 2 | _ :=
            yield 3 | x := x
```

We solve this by introducing something analogous to do-notation in Haskell. `x <- expr; y` means `expr | x := y`. We can also ignore the result: `x; y` means `x | _ := y`. Using this new notation, `countToThree` becomes:

```
countToThree : Generator Nat |= Unit := do
    yield 1
    yield 2
    yield 3
    {}
```

One last issue to solve is about running one computation inside another. Using our new notation, it is enough to provide the following function:

```
run : (e >= m) => (e |= a) -> (a -> e |= b) % m -> e |= b
```

Note the new kind of multiplicity constraint: `e >= m` for effect `e` means "`m <= n` for all multiplicities `n` in `e`". Desugaring is similar to what is done with `a <= m` constraints, only here we use `\/` instead of `/\`.

4.3.3 Handlers

After desugaring, effectful computations become recursive open unions. Therefore handlers have to be recursive functions on such unions:

```
collect : (Generator Nat & e |= Unit)
        -> State (List ! Nat) & e |= Unit :=
    | pure {} := pure {}
    | yield x cont := do
        modifyState | xs := Cons (x, xs)
        collect (cont {})
    | *other x cont := other x | y := collect (cont y)
```

However, catch-all case is the same for every effect-polymorphic handler. We can let programmers omit it. Same for `pure` cases in handlers that do not change the result of the computation.

While we're at it, let's write a `run` function from previous section.

```
run : (e >= m) => (e |= a) -> (a -> e |= b) % m -> e |= b :=
    | pure x, f := f x
    | *rest x k, f := rest x | y := run (k y) f
```

Now, the purpose of the constraint becomes clear.

4.4 Bidirectional type checking

Although we managed to simplify type annotations by employing elision rules, kind inference and implicit quantification, we still want to minify the amount of types programmer has to write out. This is the main task of type inference; two extremes of type inference landscape include:

1. Global type inference in Hindley-Milner style type systems, where the whole program can be written without a single type annotation [17];

2. Extremely local type inference, where only the most obvious annotations can be omit [19].

In our case, the necessity to only annotate top-level function declarations seems plausible; in other words, we want to annotate introduction forms, but not elimination forms. This is really close to the approach of bidirectional type checking [20] in which we split one typing judgement into two: a type *synthesis* judgement and type *checking* judgement. Care must be taken while splitting the judgement; we also have to account for implicit quantification because it makes implicit instantiation necessary. We solve the last problem by utilising Quick Look [21], an impredicative type inference engine, and extend it with rules regarding algebraic data types and effects which are fairly simple and will be given in a final version of the thesis.

4.5 Future extensions

As of now, the surface language lacks two common features of pure functional languages: nominal types and typeclasses. Both are left out of scope because they are mostly unrelated to the algebraic effects as we approached it. In addition, nominal types do not pose a significant challenge. On the other hand, typeclasses in a substructural type system face new problems. For example, consider a conventional Functor typeclass:

```
class Functor f where
    fmap : (a -> b) -> f a -> f b
```

First of all, `f` has to be a functor from types to types, not from pretypes to pretypes. This limits a space of possible instances. Furthermore, an `a -> b` argument's multiplicity is assumed to be `*`, however certain data types (`Maybe`, `NonEmpty` in Haskell) can assume stricter multiplicities. Therefore, we either have to have four similar typeclasses or introduce one two-param type class. Either decision motivates us to design a novel typeclass hierarchy which is way out of scope for this paper, although it is tackled in [22].

Chapter 5

Conclusion

As a result of the practice, we came up with the design of a surface language with algebraic effects and handlers which nicely fit into a substructural type system. This language features effect, row and multiplicity polymorphism and is suitable for modeling complex effectful interactions. All related algorithms are elaborated and are ready to be implemented in an interpreter in Haskell which will be presented as the main result of our thesis.

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.

Bibliography

[1] Unison Computing, "The Unison language," 2022, last accessed 26 April 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 26 April 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 26 April 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, "Abilities and ability handlers. Unison programming language," 2022, last accessed 26 April 2022. [Online]. Available: https://www.unison-lang.org/learn/language-reference/abilities-and-ability-handlers/#pattern-matching-on-ability-constructors

[11] D. Leijen, "The Koka programming language. Effect handlers," 2022, last accessed 26 April 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] C. A. Stone and A. P. Schoonmaker, "Equational theories with recursive types," Harvey Mudd College, Tech. Rep., 2005.

[16] S. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.

[17] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[18] "Lifetime elision - The Rust Reference," 2021, last accessed 26 April 2022. [Online]. Available: https://doc.rust-lang.org/reference/lifetime-elision.html

[19] "A Tour of Go. Type inference," last accessed 26 April 2022. [Online]. Available: https://go.dev/tour/basics/14

[20] J. Dunfield and N. Krishnaswami, "Bidirectional typing," *ACM Comput. Surv.*, vol. 54, no. 5, may 2021.

[21] A. Serrano, J. Hage, S. Peyton Jones, and D. Vytiniotis, "A quick look at impredicativity," in *International Conference on Functional Programming (ICFP'20)*, ACM. ACM, August 2020.

[22] "linear-base: Standard library for linear types," last accessed 26 April 2022. [Online]. Available: https://hackage.haskell.org/package/linear-base