Federal State Autonomous Educational Institution for Higher Education
National Research University Higher School of Economics

Faculty of Computer Science
Educational Program
Applied Mathematics and Information Science

BACHELOR'S THESIS
Program project

A New Functional Language with Effect Types and Substructural Handlers

Prepared by the student of group 183, 4th year of study,
Sokolov Pavel Pavlovich

Supervisor:
Assoc. Prof., C. Sc., **Kuznetsov Stepan Lvovich**

Moscow 2022

CONTENTS

Abstract

We propose a new programming language, OdLang, 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 implement an interpreter in Haskell. We consider the latter to be one of the main inspirations for this project.

Мы предлагаем новый язык программирования, OdLang, в котором алгебраические эффекты отслеживаются системой типов в форме полиморфных типов эффектов. Субструктурная система типов гарантирует предсказуемое поведение вычислений с эффектами; мы моделируем их, используя строковый полиморфизм и рекурсивные типы данных. В результате преобразования мы можем представить программы с эффектами в стиле передачи продолжений, который должен быть знаком работавшим с монадой `Freer`. В качестве доказательства концепции мы реализуем интерпретатор нашего языка на языке Haskell. Последний мы считаем одним из основных вдохновений для проекта.

Keywords: algebraic effects, substructural type system, bidirectional type checking, row polymorphism, equirecursive types.

CHAPTER 1

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 the topic.

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 and show the necessity to split the language in two layers — core language and surface language. In what fol-

lows, we provide a complete description of the type system, differences between surface and core languages and, finally, all related algorithmic procedures.

The results are twofold. First, we present a profound design of a sound substructural type system for a functional language with algebraic effects. Second, we provide a proof-of-concept interpreter of a core language in Haskell. Interpreter of a surface language, formal verification of soundness and implementation of efficient self-hosting compiler are reserved for future work.

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, the type system described in [14], for language with algebraic effects.

Now, let's take a step back and 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 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.

Conclusion

Our main goal for this project is to design a substructural type system capable of effect polymorphism from scratch. However, thanks to previous research, we have basic building blocks; our task is to properly fit them together.

CHAPTER 3

ODLANG CORE

This chapter is solely devoted to elaborating on how to fit together a substructural type system, row polymorphism and equirecursive types. This includes a complete description of logical rules for the core language along with the nuts and bolts of their actual implementation.

3.1 Kinds

OdLang Core is a dialect of System F_ω. It has six kinds:

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

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

Terms of kind **mult** (we call them "modalities" or "multiplicities") represent substructural restrictions we talked about. Types are data annotated with modality. Data actually represents varieties of terms — be it algebraic data types or function types.

Terms of kind **row** k are labeled collections of entries of kind k. Note that, usually, rows are collections of types; we extend the definition for a reason which will become clear once we discuss effect types in the next chapter.

$\times$ is a kind constructor for type-level pairs which are extremely useful for both mutually recursive types and desugaring of effect types. $(\to)$ is a kind constructor for kinds of type operators.

3.2 Type-level operations

To encode relations between kinds, OdLang Core has a dedicated dialect of simply typed lambda calculus with pairs and a fixpoint. Its kinding rules are fairly standard and are given in 3.1; K is a standard context supporting exchange, weakening and contraction. However, in addition to these generic rules, each kind has its own additional set of rules. To denote this, we reuse the terms provided here later and refer to them as "E-terms".

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

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

$$\frac{K\ \vdash\ a :: k \qquad K\ \vdash\ b :: k'}{K\ \vdash\ \langle a, b\rangle :: k \times k'}\text{KPair} \qquad \frac{K\ \vdash\ p\ ::\ k_1 \times k_2}{K\ \vdash\ \pi_i\ p :: k_i}\text{KProj}_i$$

$$\frac{k \text{ simple kind} \qquad K,\ a :: k\ \vdash\ b :: k}{K\ \vdash\ (\mu\ a :: k.\ b) :: k}\text{KFix} \qquad \frac{K\ \vdash\ t :: \text{type}}{K\ \vdash\ \text{mul}\ t :: \text{mult}}\text{KMul}$$

$$\frac{K\ \vdash\ t :: \text{type}}{K\ \vdash\ \text{dat}\ t :: \text{data}}\text{KDat}$$

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

Figure 3.1: Type-level operations

The most interesting bit is a new rule, KMap, which allows us to apply an operator of kind $k \to k'$ not only to terms of kind k, but also to rows of kind **row** k (elementwise). This makes row a type-level endofunctor.

Once we introduce type-level operations, we also have to determine evaluation rules and sound equivalence relations which respect evaluation. The

latter is the whole point of [16]; we only have to extend algorithms to handle polymorphic multiplicities and rows which are elaborated in the corresponding sections.

3.3 Multiplicities

Multiplicities form a distributive lattice on four elements:

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

Multiplicity literals draw an analogy with the notation of regular expressions:

- **U**nrestricted ($*$) — values of such types can be left unused and can be arbitrarily copied;
- **R**elevant ($+$) — cannot be left unused;
- **A**ffine ($?$) — cannot be copied, but can be left unused;
- **L**inear ($!$) — must be used exactly once.

$\vee$ 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:

$$[m \leqslant +] \to a^m \to (a^m, a^m) \quad \rightsquigarrow \\ \big(\text{Forall}(a :: \text{data})(m :: \text{mult}). \\ (a^{m \wedge +} \to \{.0 : a^{m \wedge +}, .1 : a^{m \wedge +}\})^*\big)^* \quad (3.1)$$

3.3.1 Multiplicity equality problem

In three following subproblems, we assume that E-terms are opaque, that is, they are treated as completely independent variables which can be assigned independent values. However, this is not always true: $\pi_1\langle *, t\rangle$ obviously reduces

to $*$ which cannot be assigned to any value other than $*$. To fix this, we first eagerly evaluate E-terms with every reduction rule except for μ-expanding. For now, we assume that this evaluation strategy yields opaque E-terms (given they are of kind **mult**).

1. MULTIPLICITY EQUALITY.

   Input: two multiplicity terms consisting of multiplicity literals, $\vee$, $\wedge$ and opaque E-terms. Assume that there is a binary predicate $a =_E b$ on E-terms which decides their equality.

   Expected output: let us call a function from E-terms to multiplicity literals a *valuation* if it respects $(=_E)$. If given multiplicity terms evaluate to the same multiplicity literal under any possible valuation, say "YES". If there exists a valuation under which the evaluation differs, output it or any subset (partial map) sufficient to show that evaluation differs.

   Solution: note that we can represent modalities as pairs of booleans (w, c), where w stands for "is weakening (drop) forbidden" and c means the same for contraction (copying). Also note that join and meet respect this representation, that is:

   $$(w, c) \wedge (w', c') = (w \wedge w', c \wedge c');$$
   $$(w, c) \vee (w', c') = (w \vee w', c \vee c').$$

   Therefore, we can check equality of multiplicity terms in the same way we could check equality of two boolean formulas formed out of $\wedge$, $\vee$ and out of E-terms as variables. This way, an output to the problem is formed as follows: check the equality of boolean formulas for the w part; if there exists a valuation from E-terms to booleans which disproves equality of booleans, we can safely output it as a partial valuation which disproves equality of modalities (c components do not matter). Otherwise check the equality for the c part.

2. EQUALITY OF MONOTONIC BOOLEAN FORMULAE.

   Input: two boolean formulas ϕ, ψ consisting of boolean literals, $\vee$, $\wedge$ and opaque E-terms. $(=_E)$ is still available.

   Expected output: a partial valuation to booleans which disproves the supposed equality of formulas, if any.

   Solution: In other words, we want to check if the following is true:

$$\forall a : (\phi(a) \equiv \psi(a))$$

   However, this first-order formula is equivalent to:

$$(\forall a : \phi(a) \to \psi(a)) \wedge (\forall a : \psi(a) \to \phi(a))$$

   We see that, to find an offending valuation, it is enough to find a valuation which offends any one of the subsumptions.

3. SUBSUMPTION OF MONOTONIC BOOLEAN FORMULAE.

   Input: two monotonic boolean formulas ϕ, ψ where variables can be tested for equality.

   Expected output: a partial valuation a which offends $\phi(a) \to \psi(a)$, if any.

   Solution: We do not know whether this problem has a polynomial solution or if it is NP-complete (it is in NP, though, because it obviously reduces to SAT). The solution we provide is worst-time exponential; it is however not of great concern because we do not expect multiplicity terms to be big.

   First of all, note that for every boolean formula we receive, we can build an equivalent CNF and DNF using a depth-first tree traversal. In addition, the following tautologies are true:

$$\forall a : \beta(a) \to (\gamma(a) \wedge \delta(a)) \equiv (\forall a : \beta(a) \to \gamma(a)) \wedge (\forall a : \beta(a) \to \delta(a));$$

$$\forall a : (\beta(a) \vee \gamma(a)) \to \delta(a) \equiv (\forall a : \beta(a) \to \delta(a)) \wedge (\forall a : \gamma(a) \to \delta(a)).$$

   Then, if we want to check if $\phi \to \psi$, it is enough to:

(a) Build a DNF of ϕ;

(b) Build a CNF of ψ;

(c) For every conjunct $K = k_1 \wedge \ldots \wedge k_m$ of ϕ and for every disjunct $L = l_1 \vee \ldots \vee l_n$ of ψ, try to find an offending valuation for $K \to L$.

But how do we find it? Consider the following: $K \to L$ is wrong iff every variable in K is set to true, but every variable in L is set to false. Therefore such a valuation exists iff variable sets of K and L are disjoint. This can be tested trivially in $O(|K| \cdot |L|)$ time.

This is worst-time exponential because both DNF and CNF can be exponentially big in comparison with the source formula.

3.4 Row types

Rows form a join semilattice of dictionaries:

$$R_k ::= \quad E \mid \varnothing_k \mid (.n : t) \mid R_k \vee R_k$$

The underscore k means that R_k describes terms of kind "row k" (therefore term t above is expected to have kind k). n is an entry label.

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.

This way, the semantic of a row is best represented as a multimap, and row equivalence is an equivalence of multimap formulas.

3.4.1 Row equality problem

Once again, we first make E-terms opaque by eagerly evaluating them safe for μ unfolding. Here, opaqueness means two things. First of all, opaque E-terms of row kind cannot reduce into $(.n : t) \vee \ldots$. Secondly, for every pair of different opaque E-terms there exists a substitution of variables that makes them reduce into different rows.

1. ROW EQUALITY.

   Input: two formulas consisting of empty sets, entry literals, $\vee$ and opaque E-terms. Assume that there is a binary predicate $a =_E b$ on E-terms which decides their equality and a binary predicate $c =_k d$ on entry values which decides their equality.

   Expected output: if given formulas evaluate to the same multimap under any possible valuation, say "YES". If there exists a valuation under which the evaluation differs, say "NO".

   Solution: row formulas admit a more compact representation which is a pair: the first component collects entries in a multimap, the second collects E-terms in a list. Once again, we can build this representation out of a formula using a depth-first tree traversal.

   The equality of a pair is an equality of its components; multimaps `a`, `b` are equal iff sets of labels are equal and, for every label `l`, lists `a[l]` and `b[l]` are equal. Therefore we reduce a ROW EQUALITY to the BAG EQUALITY of multimap entries and of lists of E-terms.

2. BAG EQUALITY.

   Input: two lists l, r of opaque values which can be tested for equality.

   Expected output: "YES" if the contents of lists are equal, "NO" otherwise.

   Solution: equality is an equivalence relation, therefore we can say that we are given a graph which is reflexively, symmetrically and transitively closed; we are tasked with finding the perfect matching in its bipartite subgraph.

   First of all, such a graph consists of disjoint cliques. Then a perfect matching exists iff a separation into bipartite graph perfectly cuts each clique in two parts of equal size.

   In conclusion, it is enough to:

   (a) Collect contents of l in a list of cliques in $O(|l|^2)$;

   (b) Do the same for r;

(c) Check that the amount of cliques are equal;

(d) Check that for every clique in l, there exists its sibling in r of the same size.

Sadly, the error reporting facilities of row equality are much worse that those of multiplicity equality: if two rows are not equal, we can only say which two bags are not equal; if bags are not equal, it means that there is no perfect matching, and there is no useful description of this fact which can be shown to the programmer.

3.5 Data

Data includes 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.

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

To elaborate on the algebraic data type semantics:

- $\{\ldots R\}$ 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.

Data equality checking algorithm is straightforwardly imported from [16].

3.6 Type-level evaluation

In the previous sections, we talked about eager evaluation of terms. Now that we have seen all the syntactic forms in the type level of our language, we can define what it means. We define it via small-step operational semantics. Its logical rules are mostly standard so in Fig.3.2 we provide only the ones involving KMul and KMap rules.

Note that we had to add functor laws for rows: it is necessary in order to preserve soundness of the algorithmic equality. Also note that making equality and evaluation mutually recursive is not a problem because they still terminate.

$$\frac{f = (a :: k.a)}{f \mathbin{@} r \rightsquigarrow r}\textsc{OpMapID}$$

$$\frac{K \vdash g :: k \to k'}{f \mathbin{@} (g \mathbin{@} r) \rightsquigarrow (a :: k.f(g\ a)) \mathbin{@} r}\textsc{OpMapComp}$$

$$\frac{K \vdash t :: k \to k'}{t \mathbin{@} \varnothing_k \rightsquigarrow \varnothing_{k'}}\textsc{OpMapEmpty}$$

$$\frac{}{f \mathbin{@} (.n : x) \rightsquigarrow (.n : fx)}\textsc{OpMapEntry}$$

$$\frac{}{f \mathbin{@} (r \vee r') \rightsquigarrow (f \mathbin{@} r) \vee (f \mathbin{@} r')}\textsc{OpMapJoin}$$

$$\frac{}{\text{mul}\ a^m \rightsquigarrow m}\textsc{OpRunMul}$$

Figure 3.2: Small-step operational semantics of type-level term evaluation

3.7 Terms

Declarative rules for terms follow straightforwardly from rules for data and are provided in Fig.3.3-3.4. Declarative rules for substructural term context are

conventional and are provided in Fig.3.3. Note that while defining operations on rows, we have to account for polymorphic rows. We do this by introducing a special entities, called **keys** of rows. They include not only names of entries, but also aliases for E-terms; to fully formalize this, we would have to introduce a third context, context of keys and related existential variables, into the typing judgement; however, this brings little explanatory power so we leave the explanation textual.

Now, how many keys a row has? One for each unique field name and one if opaque E-terms are present. In addition, we define a row access operation $R[k]$, which semantically returns a value associated with k in a row R. It works in the following way:

- if key is an entry field, and entry with such field is unique, return the associated value;
- if key is an entry field, but such entry is not unique, error out;
- if key corresponds to E-terms, return the "rounding" of the types of values in these terms. Rounding is a conservative estimation of types a row contains which uses existential variables:

$$\begin{aligned} \lfloor f \ @ \ r \rfloor &::= \ f\lfloor r \rfloor \\ \lfloor \rho_k \rfloor &::= \ \alpha(k) \end{aligned}$$

  ρ_k is an opaque E-term of kind **row** k which is not a @-term. $\alpha(k)$ is a term of kind k which uses only KPair rule and existential variables.

  The next logical step would be to try to unify the obtained estimations from all the E-terms. As we will see later, we do not need this facility to support effect polymorphism; however it would be a good idea to add the notion of "most specific unifier" later.

  One thing to note: while a type contains existential variables, terms of this type can only be used where these variables are well-scoped. Here, a variable is well-scoped if the term is used as a value with a key which was associated with a E-term.

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

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

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

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

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

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

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

Figure 3.3: System-F Fragment and Substructural Context

Another important bit is a concept of tunneling which is originally described in [19]; while we do not use a dedicated runtime mechanisms for running algebraic effects, we still share the same issues while dealing with polymorphic rows, e.g. wrong **let** unpacking, **case** mishandling etc. While we have not yet figured out the way to bypass this problem, it is our top priority for the future.

3.7.1 Context splitting

An important problem in implementing the substructural type system is a problem of context splitting. It is already apparent in the declarative rules in

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

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

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

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

$$\frac{K; \Gamma \ \vdash \ e : R[n] \qquad K \ \vdash \ \text{mul } R[n] \leqslant m}{K; \Gamma \ \vdash \ .n \ e : (|\ldots R|)^m}\text{TOrI}$$

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

Figure 3.4: Terms of Algebraic Data Types

Fig.3.3-3.4. For example, while synthesizing the type of a result of a function application, how do we split an input context into Γ and Δ? In our interpreter, we approach this by storing a description of context split in every term that requires context splitting for its corresponding logical rule. This description is a list of directions: for every variable instance in a context, there is a direction showing it in which part of term to go. It can be built by simple depth-first tree traversal.

However, this approach has several limitations:

- In TWithI and TOrE, the number of usages of a variable should be the same across all branches;
- In TAbs, the number of usages of a variable inside the function body should already be known.

We overcome this by a) storing the number of usages at function abstraction; b) generating useless applications inside branches. For example, instead of $[a : \{p : x, q : x\}, b : x]$, we do $[a : (\lambda y.\{p : y, q : y\})x, b : x]$.

CHAPTER 4

SURFACE LANGUAGE

While we do not present an interpreter of a surface language, we still provide an extensible description of its design along with schemas of all necessary algorithms.

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 * data * 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 * data * 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. Inability to define recursive types is awkward (recall that we only have recursive data). But we can provide a limited form of a type recursion, the one where multiplicity of a recursed type does not depend on itself. To desugar a recursive tuple with types, we at first replace every occurence of recursed type t with $(\text{dat } t)^{\text{mul } t}$. Then we replace every **mul** with its value which is expected after the unfolding and try to pull it outside of μ (here is why independence from itself is important). Finally, replace every **dat** with a fresh variable and erase the modality from the corresponding value in the body of recursion.

   An example should make procedure more clear:

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

tuple <Red', Blue'> : type * type :=
    (<<dr, db>, <mr, mb>> => <dr % mr, db % mb>)
    <fix (<r, b> : data * data . <
        { head: Int*, tail: (() -> b+)! },
        { head: Int*, tail: (() -> r*)? }
    >), <*, +>>
```

3. 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 [20].

Before desugaring:

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

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

After desugaring:

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

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

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

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

   To perform said kind inference, one can utilize a standard unification algorithm [21].

5. 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. However, the ordering of forall binders becomes arbitrary; to compensate for that, we provide a way to specify a named type argument:

```
const : (a -> (b -> a) % mul a)* :=
    | x, y => x
```

```
type Id := forall a. (a -> a)*

ok : Int := const @{b: Id} 5 (| x => x)
```

6. 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 [22]. 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 \/ mul a \/ mul b)
    ) % mul a)*
)*)*)* :=
    ...
```

7. 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.

8. 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

data 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 \/ mul a)
|) % (m \/ mul 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 \/ mul to)) % m) % (m \/ mul from)
```

```
data (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 => pure 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
    pure {}
```

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 "`n >= m` 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 a => pure a
    | 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 [21];

2. Extremely local type inference, where only the most obvious annotations can be omit [23].

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 [24] 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.

At first, we wanted to adapt the Quick Look inference engine [25] for impredicative polymorphism; however, after witnessing the backlash [26] in the enterprise Haskell community because of the way it interferes with eta-expansion, we decided to take a step back and employ predicative polymorphism, introducing named type application for the rare complex cases where impredicativity is required.

Although we settled for predicative polymorphism, we still want to preserve higher-kindedness; lucky for us, exactly this kind of type system is bidirectionally type checked in a beautiful paper by J.Dunfield and N.R.Krishnaswami [27]. To extend it, we need to both add new checking/synthesis rules and extend algorithmic subtyping to solve equations with rows and multiplicities. The most interesting of the new bidirectional rules are provided in Fig.4.1; note that we do not check multiplicities as it will be done during the checking phase in the Core.

Stated formally, a multiplicity equation problem sounds like: given a system of equalities on multiplicity formulas with variables in domains X and Y, find any substitution for X that system is satisfied under any substitution of Y. Row equation problem sounds exactly the same. The main difference is that while row equation problem could be solved as if it was a system of linear equations, a multiplicity equation problem does not have a simple solution.

$$\frac{\forall n \in \text{keys } R : \Psi \vdash e_n \Rightarrow R[n]}{\Psi \vdash \{\overline{.n : e_n}\} \Rightarrow \Pi(m :: \text{mult}).\{\ldots R\}^{m \vee \sup R}}\textsc{DeclAndI}\Rightarrow$$

$$\frac{\Psi \vdash w \Leftarrow [.n : t,\ \ldots R]^m}{\Psi \vdash w.n \Leftarrow t}\textsc{DeclWithE}\Leftarrow$$

$$\frac{\Psi \vdash w \Leftarrow \{.n : t,\ \ldots R\}^m}{\Psi \vdash w.n \Leftarrow t}\textsc{DeclPun}\Leftarrow$$

$$\frac{\Psi \vdash e \Rightarrow t}{\Psi \vdash .n\ e \Rightarrow \Pi\, r\, m.(|.n : t, \ldots r|)^{m \vee \text{mul}\, t}}\textsc{DeclOrI}\Rightarrow$$

Figure 4.1: Bidirectional rules for Algebraic Data Types

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 data to data. 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 [28].

CHAPTER 5

CONCLUSION

First of all, 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 etc. In addition, we saved ourselves the headache of developing a dedicated runtime for algebraic effects using a well-typed encoding into continuation-passing style.

Secondly, we have implemented an interpreter of the core language described above except for problems with tunneling whose resolution is our primary goal after the project. Source code is available at `https://github.com/TurtlePU/odlang`.

Last but not least, 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 row, effect 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.

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; and so forth. To overcome these limitations in the future, 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 subsequent goals after

this project are to write an interpreter for the surface language, formally verify soundness of the type system and write a self-hosting compiler. Formal verification 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.

BIBLIOGRAPHY

[1] Unison Computing, "The Unison language," 2022, last accessed 13 May 2022. [Online]. Available: https://www.unison-lang.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 13 May 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 13 May 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 13 May 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 13 May 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.

[19] Y. Zhang and A. C. Myers, "Abstraction-safe effect handlers via tunneling," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

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

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

[22] "Lifetime elision - The Rust Reference," 2021, last accessed 13 May 2022. [Online]. Available: https://doc.rust-lang.org/reference/lifetime-elision.html

[23] "A Tour of Go. Type inference," last accessed 13 May 2022. [Online]. Available: https://go.dev/tour/basics/14

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

[25] 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.

[26] "Was simplified subsumption worth it for industry haskell programmers?" last accessed 13 May 2022. [Online]. Available: https://www.reddit.com/r/haskell/comments/ujpzx3/was_simplified_subsumption_worth_it_for_industry

[27] J. Dunfield and N. Krishnaswami, "Complete and easy bidirectional typechecking for higher-rank polymorphism," *ACM SIGPLAN Notices*, vol. 48, 06 2013.

[28] "linear-base: Standard library for linear types," last accessed 13 May 2022. [Online]. Available: https://hackage.haskell.org/package/linear-base