Can functional programming be liberated from the natural deduction style?

Programming in the Sequent Calculus

David Binder / Proofs, Computation and Meaning / 2022 Online

How does the choice of a logical calculus influence programming style?

We shall use Church's method for denoting definitions (see Church 1932, p. 355) and shall list the following, giving on the right the equivalent in Church's notation:

$T \to JII$	$\lambda xf \cdot f(x)$	(See footnote 4)
$C \to JT(JT)(JT)$	$\lambda fxy \cdot f(y, x)$	
$B \to C(JIC)(JI)$	$\lambda fgx \cdot f(g(x))$	
$W \to C(C(BC(C(BJT)T))T)$	$\lambda fx \cdot f(x, x)$	
$1 \to BI$	$\lambda fx \cdot f(x)$	
$\mathfrak{p} \times \mathfrak{q} \to B\mathfrak{pq}$	$\lambda x \cdot \mathfrak{p}(\mathfrak{q}(x))$	

[4] $\lambda xf \cdot f(x)$ is an abbreviation of $\lambda x \lambda f \cdot f(x)$.

Wadler's Dual Calculus

Terms in the dual calculus are not always easy to read. Compare, for instance, the λ-calculus term

$$\langle \text{fst } V, \text{snd } M \rangle$$

with a corresponding dual calculus term,

$$(V \bullet \text{fst}[x.(V \bullet \text{snd}[y.(\gamma \bullet \langle x, y \rangle)])]).\gamma$$

The latter is reminiscent of continuation-passing style — like the Pompidou Center in Paris, the plumbing is exposed on the outside. While this can make the expression harder on the eyes, it also — like CPS, and like the Pompidou Center — has the advantage of revealing structure that previously was hidden.

Overview

1. **Current State:**
   Functional programming based on Curry-Howard for *natural deduction.*

2. **What can we hope to gain:**
   What are the limitations of (intuitionistic) natural deduction? What can the (classical) sequent calculus contribute?

3. **A Vision:**
   Programming in the sequent calculus. (With real code examples!)

Goal: Make you understand why PL researchers might be interested in using sequent calculus as a programming language.

The Current State

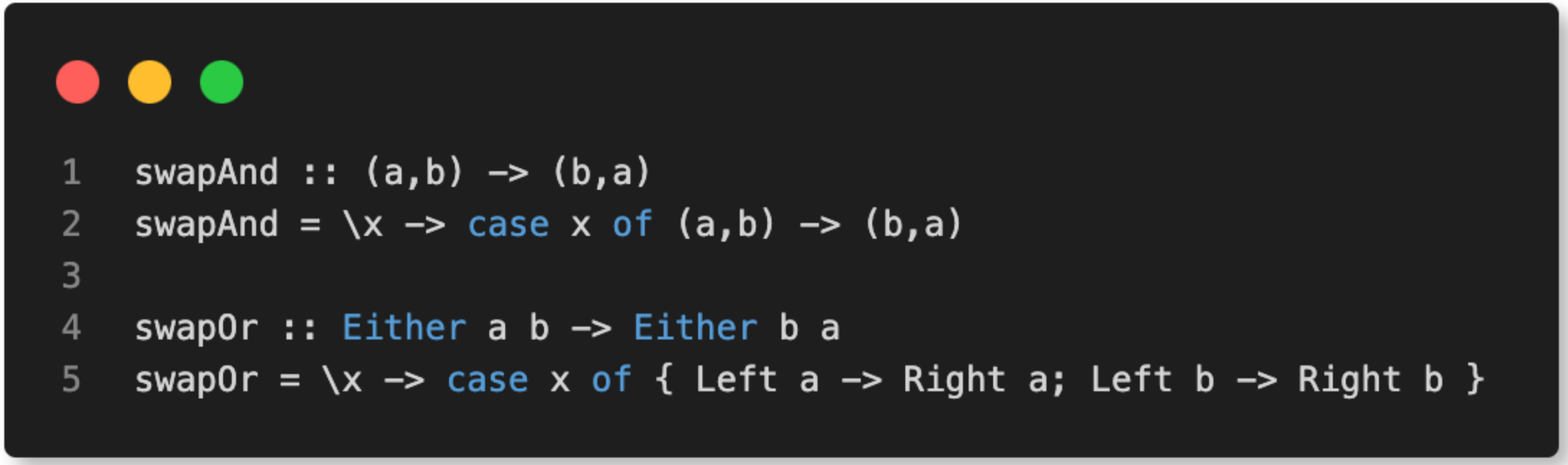
The current state

The Curry-Howard Isomorphism permeates theory and practice of FP.

- **Programmers:** Programmers use functional programming languages closely modelled upon Church's λ-calculus.
- **Compiler implementors:** Optimization and implementation are studied using intermediate representations based on, and abstract machines for the lambda calculus.
- **PL researchers:** The percentage of papers containing some version of the following rule is very high:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}$$

Programming in natural deduction style



```haskell
swapAnd :: (a,b) -> (b,a)
swapAnd = \x -> case x of (a,b) -> (b,a)

swapOr :: Either a b -> Either b a
swapOr = \x -> case x of { Left a -> Right a; Left b -> Right b }
```

What is distinctive about this programming style, i.e. intuitionistic natural deduction?

Properties of Natural Deduction

- There is only ever one formula on the right of the turnstile.
- There is only one judgement form $\Gamma \vdash t : \tau$ which types proofs of a type.
  ⇒Refutations (resp. consumers/continuations) are not first class.
- Rules only operate on the right hand side of the sequent.
  ⇒No rules for formulas on the left of the turnstile.
- Rules come in introduction (e.g. $\lambda x\,.\,e$) and elimination forms (e.g. $(f\ a)$).
- Computation happens when introduction and elimination forms interact.
  ⇒Cut is internalized.

Collectively, these properties determine the programming style!

The sequent calculus

Natural Deduction vs Sequent Calculus

ND: There is only one formula on the RHS

In natural deduction we have $\Gamma \vdash \phi$, in sequent calculus $\Gamma \vdash \Delta$.

- There is more than one point that we can return to.
- Alternatively: We have multiple continuations in context.

```
1  public static int foo(float x, String y) throws IOException {
2      ...
3  }
```

We should read this Java example as: float, String $\vdash$ int, IOException

Natural Deduction vs. Sequent Calculus

ND: There is only one judgement form

- A judgement for well-typed proofs/producers: $\Gamma \vdash e : \phi \mid \Delta$
- A judgement for well-typed refutations/consumers: $\Gamma \mid e : \phi \vdash \Delta$
- A judgement for well-typed commands: $c : (\Gamma \vdash \Delta)$

Natural Deduction vs. Sequent Calculus

ND: Rules only operate on the RHS.

$$\frac{\forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\Gamma \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : T_1 \oplus T_2 \vdash \Delta}\ (\text{L-}\oplus\text{-Intro})$$

$$\frac{\Gamma \vdash e : T_i \mid \Delta}{\Gamma \vdash \mathbf{In}_i\ e : T_1 \oplus T_2 \mid \Delta}\ (\text{R-}\oplus\text{-Intro}_i)$$

$$\frac{\Gamma \mid f : T_1 \oplus T_2 \vdash \Delta}{\Gamma \mid \mathbf{Out}_i\ f : T_i \vdash \Delta}\ (\text{L-}\oplus\text{-Elim}_i)$$

$$\frac{\Gamma \vdash e : T_1 \oplus T_2 \mid \Delta \qquad \forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : (\Gamma \vdash \Delta)}\ (\text{R-}\oplus\text{-Elim})$$

Natural Deduction vs. Sequent Calculus

ND: Computation happens where introduction and elimination meet.

In sequent calculus, computation happens at a cut, where a producer and a consumer of the same type meet.

$$\frac{\Gamma \vdash e : T \mid \Delta \qquad \Gamma \mid f : T \vdash \Delta}{\langle e \parallel f \rangle : \Gamma \vdash \Delta}\ (\textsc{Cut})$$

```
1   def prd foo : T := ...;
2   def cns bar : T := ...;
3   def cmd biz := foo >> bar;
4
```

RQ1: Evaluation Order

- Sequent calculus is an excellent tool for studying evaluation orders.
- The critical pair $\langle \mu x \,.\, c_1 \mid \tilde{\mu}\alpha \,.\, c_2 \rangle$ can be reduced in two different ways.
- Call-by-value: Reduce the μ-abstraction.
- Call-by-name: Reduce the $\tilde{\mu}$-abstraction.
- Mixing evaluation orders using Shift connectives.
- In sequent calculus, call-by-need dualizes to call-by-coneed. Useful?

RQ2: Data vs. Codata

- Data and Codata types are the two ways to introduce new types.
- Corresponds to polarity in linear logic.
- In sequent calculus, data and codata types are more symmetric:
- ND: Constructors are first class, destructors are not first class.
- ND: Destructors of codata types have one "output" type. Constructors of data types don't.
- ND: We cannot define all the connectives from linear logic.

RQ3: Sequent Calculus as an Intermediate Language

Sequent Calculus as a Compiler Intermediate Language

Paul Downen Luke Maurer
Zena M. Ariola
University of Oregon, USA
{pdownen,maurerl,ariola}@cs.uoregon.edu

Simon Peyton Jones
Microsoft Research Cambridge, UK
simonpj@microsoft.com

- Reduction in sequent calculus is already like in an abstract machine.

$$\begin{array}{lcl}
\langle \lambda(x \cdot \alpha).c \mid\mid v \cdot f \rangle & \triangleright_\beta & c\{x := v, \alpha := f\} \\
\langle\, \mathbf{In}_j\ v \mid\mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\}\, \rangle & \triangleright_\beta & c_j\{x_j := v\} \\
\langle [v_1, v_2] \mid\mid \mathbf{Match}\ \{[x, y] \mapsto c\} \rangle & \triangleright_\beta & c\{x := v_1, y := v_2\}
\end{array}$$

- Many optimizations (e.g. case-of-case) are simple reductions.
- Helped discover the concept of join points ($\sim \phi$-nodes).
- Target language for language features which need first-class continuations (effect systems, coroutines, exceptions).

RQ4: What algorithms are naturally expressed in SQ

- The operational essence of classical logic are control operators: a functional goto.
- There are two ways to write programs using control effects: control operators or in continuation-passing-style (CPS).
- Both ways are difficult to read and to reason about.
- Can we express these algorithms more modularily, comprehensibly, extensibly using sequent calculus?

Programming in the Sequent Calculus

Positive Products

$$\frac{\Gamma \vdash e : T_1 \otimes T_2 \mid \Delta \quad c : (x : T_1, y : T_2, \Gamma \vdash \Delta)}{\textbf{Case}\ e\ \{[x, y] \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{R-}\otimes\text{-Elim})$$

$$\frac{c : (x : T_1, y : T_2, \Gamma \vdash \Delta)}{\Gamma \mid \textbf{Match}\ \{[x, y] \mapsto c\} : T_1 \otimes T_2 \vdash \Delta}\ (\text{L-}\otimes\text{-Intro})$$

$$\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash \mu\alpha.c : T \mid \Delta}\ (\text{R-}\mu)$$

```
1  data Tensor : (a : *, b : *) -> * {
2      MkTensor(a, b)
3  };
4
5  def prd fst := \p => case p of { MkTensor(x,y) => x};
6  def prd snd := \p => mu k. p >> case { MkTensor(x,y) => y >> k };
```

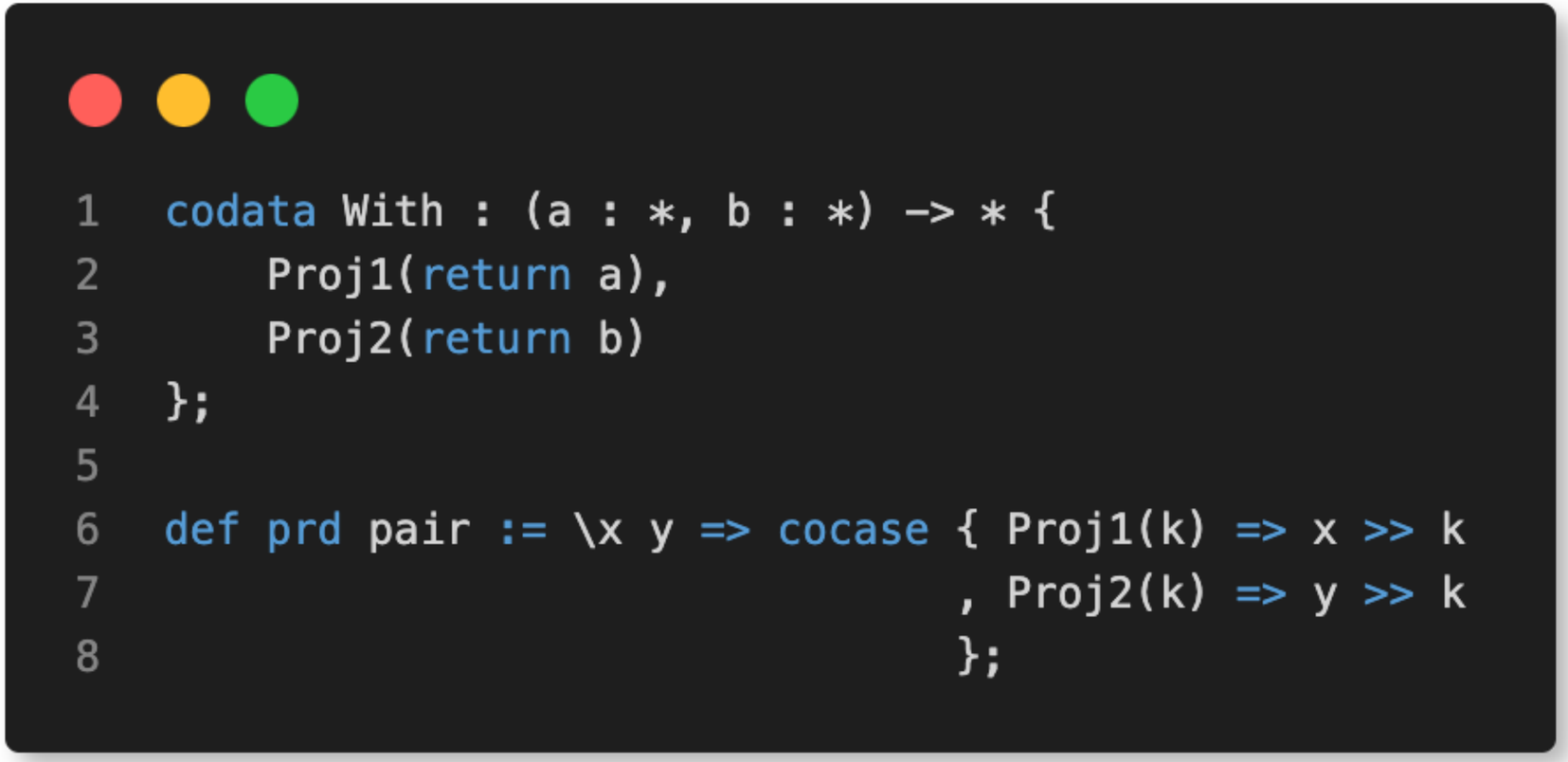
Positive Sums

$$\frac{\Gamma \vdash e : T_1 \oplus T_2 \mid \Delta \qquad \forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\textbf{Case}\ e\ \{\overline{\textbf{In}_i\ x_i \mapsto c_i}\} : (\Gamma \vdash \Delta)}\ (\text{R-}\oplus\text{-Elim})$$

$$\frac{\Gamma \vdash e : T_i \mid \Delta}{\Gamma \vdash \textbf{In}_i\ e : T_1 \oplus T_2 \mid \Delta}\ (\text{R-}\oplus\text{-Intro}_i)$$

```
1  data Plus : (a : *, b : *) -> * {
2      Left(a),
3      Right(b)
4  };
5
6  def prd swap :=
7    \x => case x of { Left(x) => Right(x)
8                    , Right(y) => Left(y)
9                    };
```

Negative Products



```
codata With : (a : *, b : *) -> * {
    Proj1(return a),
    Proj2(return b)
};

def prd pair := \x y => cocase { Proj1(k) => x >> k
                               , Proj2(k) => y >> k
                               };
```

Negative Pairs

```
codata Par : (a : *, b : *) -> * {
    MkPar(return a, return b)
};

def prd ret := \x => cocase { MkPar(k_err, k_res) => x  >> k_res };
def prd throw := \x => cocase { MkPar(k_err, k_res) => x >> k_err };
```

```
public static int foo(float x, String y) throws IOException {
    ...
}
```

Filter

```
1  -- | Filters all predicates satisfying a given predicate.
2  def rec prd filter : forall a. (a -> Bool) -> List(a) -> List(a) :=
3      \f xs => case xs of {
4          Nil => Nil,
5          Cons(y, ys) => case (f y) of {
6              True => Cons(y, filter f ys),
7              False => filter f ys
8          }
9      };
```

The implementation is not optimal. The list is copied in memory even if all elements satisfy the predicate.

Parsimonious Filter

```
1  def rec prd filterHelper := \p l => case l of {
2      Nil => cocase {
3          MkPar(k1,*) => MkUnit
4      },
5      Cons(x,xs) => case p x of {
6          True =>
7              cocase {
8                  MkPar(k1,k2) =>
9                      filterHelper p xs >> MkPar(mu ys. Cons(x,ys) >> k1, k2)
10             },
11         False =>
12             cocase {
13                 MkPar(k1,k2) =>
14                     filterHelper p xs >> MkPar(k1, mu y. xs >> k1)
15             }
16     }
17 };
18
19 def prd filter  := \p l => mu k. filterHelper p l >> MkPar(k, mu y. l >> k);
```

(Cp. Shivers & Fisher: Multi Return Function Calls)

A more general problem: The space of programming styles.

What if we mix rules?

$$\frac{\forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\Gamma \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : T_1 \oplus T_2 \vdash \Delta}\ (\text{L-}\oplus\text{-Intro})$$

$$\frac{\Gamma \vdash e : T_i \mid \Delta}{\Gamma \vdash \mathbf{In}_i\ e : T_1 \oplus T_2 \mid \Delta}\ (\text{R-}\oplus\text{-Intro}_i)$$

$$\frac{\Gamma \mid f : T_1 \oplus T_2 \vdash \Delta}{\Gamma \mid \mathbf{Out}_i\ f : T_i \vdash \Delta}\ (\text{L-}\oplus\text{-Elim}_i)$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_1 \oplus T_2 \mid \Delta \\ \forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)\end{array}}{\mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : (\Gamma \vdash \Delta)}\ (\text{R-}\oplus\text{-Elim})$$

Table 3. Four different ways to swap the components of $z : X \oplus Y$ and send to consumer $\alpha : Y \oplus X$.

Calculus	Program
Right	$\mathbf{Case}\ z\ \{\mathbf{In}_1\ x \mapsto \langle \mathbf{In}_2\ x \mid\mid \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle \mathbf{In}_1\ y \mid\mid \alpha\rangle\}$
Intro	$\langle z \mid\mid \mathbf{Match}\ \{\mathbf{In}_1\ x \mapsto \langle \mathbf{In}_2\ x \mid\mid \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle \mathbf{In}_1\ y \mid\mid \alpha\rangle\}\rangle$
Left	$\langle z \mid\mid \mathbf{Match}\ \{\mathbf{In}_1\ x \mapsto \langle x \mid\mid \mathbf{Out}_2\ \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle y \mid\mid \mathbf{Out}_1\ \alpha\rangle\}\rangle$
Elim	$\mathbf{Case}\ z\ \{\mathbf{In}_1\ x \mapsto \langle x \mid\mid \mathbf{Out}_2\ \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle y \mid\mid \mathbf{Out}_1\ \alpha\rangle\}$

```
1  def prd right : forall a. (Unit & a) & a -> Plus(Unit,Plus(Plus(a,Unit),Unit))  :=
2     cocase { Ap(x,k) =>  Right(Left(Left(x.Proj1(*).Proj2(*)))) >> k };
3
4  def prd intro : forall a. (Unit & a) & a -> Plus(Unit,Plus(Plus(a,Unit),Unit))  :=
5     cocase { Ap(x,k) => x >> Proj1(Proj2(mu y. Right(Left(Left(y))) >> k ))} ;
6
7  def prd left :  forall a. (Unit & a) & a -> Plus(Unit,Plus(Plus(a,Unit),Unit))  :=
8     cocase { Ap(x,k) => x >> Proj1(Proj2(Left(*) ;; Left(*) ;; Right(*) ;; k ))} ;
9
10 def prd elim  :  forall a. (Unit & a) & a -> Plus(Unit,Plus(Plus(a,Unit),Unit))  :=
11    cocase { Ap(x,k) => x.Proj1(*).Proj2(*) >> Left(*) ;; Left(*) ;; Right(*) ;; k } ;
```

More Info

- **Introduction and Elimination, Left and Right**
  dl.acm.org/doi/10.1145/3547637

- github.com/duo-lang

Introduction and Elimination, Left and Right

KLAUS OSTERMANN, University of Tübingen, Germany
DAVID BINDER, University of Tübingen, Germany
INGO SKUPIN, University of Tübingen, Germany
TIM SÜBERKRÜB, University of Tübingen, Germany
PAUL DOWNEN, University of Massachusetts Lowell, USA

10

Functional programming language design has been shaped by the framework of natural deduction, in which language constructs are divided into introduction and elimination rules for *producers* of values. In sequent calculus-based languages, left introduction rules replace (right) elimination rules and provide a dedicated sublanguage for *consumers* of values. In this paper, we present and analyze a wider design space of programming languages which encompasses four kinds of rules: Introduction and elimination, both left and right. We analyze the influence of rule choice on program structure and argue that having all kinds of rules enriches a programmer's modularity arsenal. In particular, we identify four ways of adhering to the principle that "the structure of the program follows the structure of the data" and show that they correspond to the four possible choices of rules. We also propose the principle of *bi-expressibility* to guide and validate the design of rules for a connective. Finally, we deepen the well-known dualities between different connectives by means of the proof/refutation duality.

CCS Concepts: • **Theory of computation** → *Abstract machines*; *Lambda calculus*; *Type theory*; *Proof theory*; *Linear logic*.

Additional Key Words and Phrases: Duality, Sequent Calculus, Natural Deduction

ACM Reference Format:
Klaus Ostermann, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen. 2022. Introduction and Elimination, Left and Right. *Proc. ACM Program. Lang.* 6, ICFP, Article 106 (August 2022), 28 pages. https://doi.org/10.1145/3547637

1 INTRODUCTION

Undoubtedly, the λ-calculus has had a profound impact on functional programming: from language design, to implementation, to practical programming techniques. Through the Curry-Howard correspondence, we know that the λ-calculus — and likewise, λ-based functional languages — are oriented around the interplay between the *introduction* and *elimination* rules of types as first formulated in natural deduction (ND). This natural deduction style of programming nicely allows for a quite "natural" way of combining sub-problems in programs, like basic operations of function composition $f\ (g\ x)$ and swapping the pair x as $(\mathbf{Snd}\ x, \mathbf{Fst}\ x)$. The natural compositional style is afforded by the fact that all expressions in the λ-calculus *produce* exactly one result that is implicitly taken by *exactly one* consumer: namely, the enclosing context of that expression. While the single implicit consumer is natural for composition, it can be rather *unnatural* if we ever want to work

Authors' addresses: Klaus Ostermann, University of Tübingen, Germany, klaus.ostermann@uni-tuebingen.de; David Binder, University of Tübingen, Germany, david.binder@uni-tuebingen.de; Ingo Skupin, University of Tübingen, Germany, skupin@informatik.uni-tuebingen.de; Tim Süberkrüb, University of Tübingen, Germany, tim.sueberkrueb@student.uni-tuebingen.de; Paul Downen, University of Massachusetts Lowell, USA, Paul_Downen@uml.edu.

2475-1421/2022/8-ART106
https://doi.org/10.1145/3547637

Thank you for your attention