QuB

A Resource Aware Functional Programming Language

Apoorv Ingle

The University of Kansas

Table of Contents

Hard problems in programming

Naming variables

Introduction and Motivation

Hard problems in programming

Resource management

in evolving production code

Resources: Files, database connections, entity with a shared state

Resource Management: File Handling

- Modified File Handling API in Haskell

$$
\begin{aligned}
\texttt{openFile} &:: \texttt{FilePath} \rightarrow \texttt{IO FileHandle} \\
\texttt{closeFile} &:: \texttt{FileHandle} \rightarrow \texttt{IO}\ () \\
\texttt{readLine} &:: \texttt{FileHandle} \rightarrow \texttt{IO}\ (\texttt{String}, \texttt{FileHandle}) \\
\texttt{writeFile} &:: \texttt{String} \rightarrow \texttt{FileHandle} \\
&\quad \rightarrow \texttt{IO}\ ((), \texttt{FileHandle}) \\
\\
\texttt{upper} &:: \texttt{String} \rightarrow \texttt{String}
\end{aligned}
$$

Resource Management: File Handling

- File Handling in Haskell

```
do f      ← openFile "sample.txt"
   (s, f)  ← readLine f
   let c = upper s
   ((), f) ← writeLine f c
         .
         .
         .
   ()   ← closeFile f
```

Resource Management: File Handling

- File Handling in Haskell Gone Wrong (Part I)

```
do f  ← openFile "sample.txt"
   (s, f)  ← readLine f
   let c = upper s
   ((), f) ← writeLine f c
         .
         .
         .
   () ← closeFile f
         .
         .
         .
   () ← closeFile f
   return c
```

Resource Management: File Handling

- File Handling in Haskell Gone Wrong (Part I)

```
do f ← openFile "sample.txt"
   (s, f) ← readLine f
   let c = upper s
   ((), f) ← writeLine f c
         .
         .
         .
   () ← closeFile f
         .
         .
         .
   () ← closeFile f
   return c
```

- File is closed twice: Run time crash

Resource Management: File Handling

- File Handling in Haskell Gone Wrong (Part II)

```
do f    ← openFile "sample.txt"
   (s, f)  ← readLine f
   let c = upper s
   ((), f) ← writeLine f c
       .
       .
       .
   return c
```

Resource Management: File Handling

- File Handling in Haskell Gone Wrong (Part II)

```
do f  ← openFile "sample.txt"
   (s, f)  ← readLine f
   let c = upper s
   ((), f) ← writeLine f c
       .
       .
       .
   return c {- File not closed!! -}
```

- File not closed: Memory leak

Resource Management: Exception Handling

- MonadError[6] in Haskell

```
class Monad m ⇒ MonadError e m | m → e where
    throwError :: e → m a
    catchError :: m a → (e → m a) → m a
```

- throwError starts exception processing
- catchError exception handler

Resource Management: Exception Handling

- Using `MonadError` in Haskell

```
do f ← openFile "sample.txt"
   ((s, f) ← readLine f
   let c = upper s
   () ← closeFile f
   return $ Right c)
       `catchError` (\_ →
              return $ Left "Error in reading file")
```

- Exception may cause memory leak

Well typed programs do not go wrong.

— R. Milner

Well typed programs do not go wrong.

— R. Milner

~~*Lights*~~*Types will guide you home*

— Coldplay

Contributions

- Design and implement QuB type system
  - Resources as first class citizens
  - Program objects are restricted or unrestricted
  - Functions that share resources with their arguments or are separate.
- Formalizing and proving important properties of QuB
- QuB is logic of ***BI*** on steroids
  - Typing Environments as graphs
- Working examples

Background Work: Simply Typed Lambda Calculus (STLC)

$$\lambda x.M \begin{cases} \text{Abstract over computation} \\ \text{Define functions} \end{cases}$$

$$MN \begin{cases} \text{Do the computation} \\ \text{Use functions} \end{cases}$$

$$\lambda x.M \begin{cases} \text{Abstract over computation} \\ \text{Define functions} \end{cases}$$

$$MN \begin{cases} \text{Do the computation} \\ \text{Use functions} \end{cases}$$

$$\frac{\Gamma_x, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'}\,[\to \mathrm{I}] \qquad \frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'}\,[\to \mathrm{E}]$$

Background Work: Simply Typed Lambda Calculus (STLC)

$$\lambda x.M \begin{cases} \text{Abstract over computation} \\ \text{Define functions} \end{cases}$$

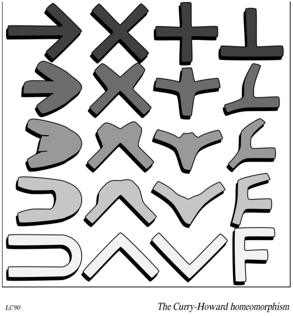
$$MN \begin{cases} \text{Do the computation} \\ \text{Use functions} \end{cases}$$

$$\frac{\Gamma_x, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'} [\to \mathrm{I}] \qquad \frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'} [\to \mathrm{E}]$$

Hindley-Milner (***HM***) type system ensures sane programs

- Types are Propostions
- Programs are Proofs

HM type system ≡ Second Order Intuitionistic Propositional Logic



Source: `http://lucacardelli.name/Artifacts/Drawings/CurryHoward/CurryHoward.pdf`

Background Work: Second Order Intuitionistic Propositional Logic

$\boxed{\text{Language}}$

$$\text{Propostions \& connectives} \quad A, B, C ::= x \mid A \supset B \mid \forall x.B \mid \ldots$$
$$\text{Context} \quad \Gamma, \Delta ::= \epsilon \mid \Gamma, A$$

$\boxed{\text{Logic Rules}}$

$$\frac{}{A \vdash A}\,[\mathrm{Ax}]$$

$$\frac{\Gamma \vdash B \qquad x \notin \Gamma}{\forall x.B}\,[\forall \mathrm{I}] \qquad \frac{\Gamma \vdash \forall x.B \qquad \Gamma \vdash A}{B[x/A]}\,[\forall \mathrm{E}]$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\,[\supset \mathrm{I}] \qquad \frac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,[\supset \mathrm{E}]$$

Background Work: Second Order Intuitionistic Propositional Logic

Propositions are truth values not resources

$\boxed{\text{Language}}$

Propostions & connectives $\quad A, B, C ::= x \mid A \supset B \mid \forall x.B \mid \ldots$

Context $\quad \Gamma, \Delta ::= \epsilon \mid \Gamma, A$

$\boxed{\text{Logic Rules}}$

$$\frac{}{A \vdash A}\,[\mathrm{Ax}]$$

$$\frac{\Gamma \vdash B \qquad x \notin \Gamma}{\forall x.B}\,[\forall \mathrm{I}] \qquad \frac{\Gamma \vdash \forall x.B \qquad \Gamma \vdash A}{B[x/A]}\,[\forall \mathrm{E}]$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\,[\supset \mathrm{I}] \qquad \frac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,[\supset \mathrm{E}]$$

Background Work: Substructural Logic

- Structural rules implicit in intuitionistic propositional logics

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ [\mathrm{WKN}] \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ [\mathrm{CTR}] \qquad \frac{\Gamma, \Delta \vdash B}{\Delta, \Gamma \vdash B}\ [\mathrm{EXCH}]$$

Background Work: Substructural Logic

- Structural rules implicit in intuitionistic propositional logics

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ [\text{WKN}] \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ [\text{CTR}] \qquad \frac{\Gamma, \Delta \vdash B}{\Delta, \Gamma \vdash B}\ [\text{EXCH}]$$

- Control the use of [WKN] and [CTR]

Propositions now behave like resources

Background Work: Substructural Logic

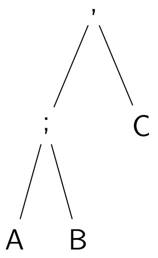
System	Who	Restrictions
Linear Logic[2]	Girard	[WKN] [CTRN]
Lambek Logic[5]	Lambek	[EXCH]
Logic of Bunched Implications[8]	O'Hearn and Pym	[WKN] [CTRN]
⋮	⋮	⋮

Background Work: Logic of Bunched Implications (***BI***)

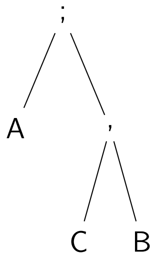
- Contexts are usually lists or sets

$$\Gamma, A, B$$

- In logic of ***BI***, contexts are trees and called are bunches
- Two connective used to combine bunches: $A; B$ or A, B



$\Gamma = (A; B), C$



$\Gamma = A; (B, C)$

Background Work: Logic of *BI*

Structural rules guided by context connectives

- Weakening

$$A \vdash A; A$$
$$A \nvdash A, A$$

- Contraction

$$A; A \vdash A \qquad A; B \vdash B$$
$$A, B \nvdash A \qquad A, B \nvdash B$$

Interpretation:

- Propositions connected with , are separate resources
- Propositions connected with ; are sharing resources

Background Work: Logic of ***BI***

(Absence of) Structural rules and logical connectives:

- Meaning of conjunction

$$A, B \vdash A \otimes B \qquad\qquad A; B \vdash A \,\&\, B$$

- Meaning of implication

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \mathrel{-\!\!*} B}\,[\mathrel{-\!\!*}\mathrm{I}] \qquad\qquad \frac{\Gamma; A \vdash B}{\Gamma \vdash A \twoheadrightarrow B}\,[\twoheadrightarrow\mathrm{I}]$$

Background work: Logic of *BI*

Coffee Shop

1 cup coffee costs \$2

$\otimes$

$\vdash$

Coffee Shop

1 cup coffee costs \$2

$\otimes$

$\vdash$

$\&$

$\nvdash$

Coffee Shop

1 cup coffee costs $2

$\vdash$

$-\!\!*$

$;$

$-\!\!*$

$\nvdash$

QuB: Curry-Howard interpretation of logic of $\boldsymbol{BI}$

QuB: Types

$$\begin{array}{ll} \text{Types} & \tau, \upsilon, \phi ::= \textit{\textbf{t}} \mid \iota \mid \tau \to \tau \\ & \qquad\qquad \text{where} \qquad \to \in \{\twoheadrightarrow\!\!\!\!\ast, \twoheadrightarrow\} \end{array}$$

- $\rightarrowtail$: Function type that is separate from its argument
- $\twoheadrightarrow$: Function type that is in sharing with its argument

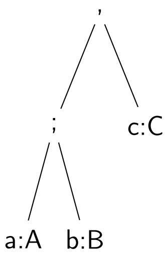
QuB: Expression Language

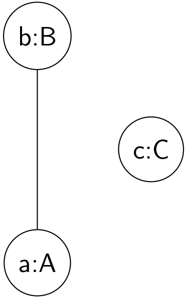
$$
\begin{aligned}
\text{Term Variables} \quad & x, y, z \in \mathrm{Var} \\
\text{Expressions} \quad & M, N ::= x \\
& \quad \mid \lambda^{-\!*} x.M \mid \lambda^{\twoheadrightarrow} x.M \\
& \quad \mid MN \mid \texttt{let } x = M \texttt{ in } N
\end{aligned}
$$

- $\lambda^{-\!*} x.M$: Argument x separate from M
- $\lambda^{\twoheadrightarrow} x.N$: Argument x sharing with M

QuB: Typing Environment

- Logic of ***BI***: Contexts are trees
- QuB: Contexts generalized to graphs



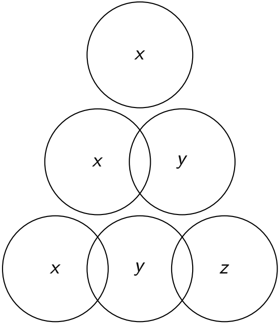


- Nodes are program objects
- (No) Edges represent (no) sharing

QuB: Typing Environment

Sharing relation Ψ

reflexive	$\forall x.\ x \,\Psi\, x$	
symmetric	$\forall x, y.\ x \,\Psi\, y \Rightarrow y \,\Psi\, x$	
non-transitive	$\forall x, y, z.\ x \,\Psi\, y \wedge y \,\Psi\, z \not\Rightarrow x \,\Psi\, z$	



QuB: Typing Environment

Adjacency lists for sharing graphs

"x of type σ is in sharing with $\vec{y}$"

$$(x, \sigma, \vec{y}) \in \Gamma$$

Typing Context $\quad \Gamma, \Delta ::= \epsilon \mid \Gamma, x^{\vec{y}} : \sigma$

Examples: Basic Data Structures

- Multiplicative Product (Separating Pair)

$$\tau \otimes \tau' = \tau \mathrel{-\!\!*} \tau' \mathrel{-\!\!*} (\tau \mathrel{-\!\!*} \tau' \mathrel{-\!\!*} \upsilon) \mathrel{-\!\!*} \upsilon$$
$$\langle x, y \rangle = \lambda^{\mathrel{-\!\!*}} x.\lambda^{\mathrel{-\!\!*}} y.\lambda^{\mathrel{-\!\!*}} f.fxy$$

- Additive Product (Sharing Pair)

$$\tau \,\&\, \tau' = \tau \mathrel{-\!\!*} \tau' \twoheadrightarrow (\tau \mathrel{-\!\!*} \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon$$
$$\langle x; y \rangle = \lambda^{\mathrel{-\!\!*}} x.\lambda^{\twoheadrightarrow} y.\lambda^{\twoheadrightarrow} f.fxy$$

- Sums

$$\tau \oplus \tau' = (\tau \mathrel{-\!\!*} \upsilon) \to (\tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon$$
$$\texttt{case}\ c\ \texttt{of}\ \{f; g\} = \lambda^{\mathrel{-\!\!*}} c.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.cfg$$

$$\begin{aligned} \texttt{inl} &: \tau \mathrel{-\!\!*} (\tau \oplus \tau') & \qquad \texttt{inr} &: \tau' \mathrel{-\!\!*} (\tau \oplus \tau') \\ \texttt{inl} &= \lambda^{\mathrel{-\!\!*}} x.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.fx & \qquad \texttt{inr} &= \lambda^{\mathrel{-\!\!*}} y.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.gy \end{aligned}$$

Programmer Friendly

- Define custom types

```
data Bool   = True | False
data List a = Nil | Cons a a
data Tree a = Leaf | Node a a
```

$$\vdots$$

Programmer Friendly

- Define custom types

```
data Bool   = True | False
data List a = Nil | Cons a a
data Tree a = Leaf | Node a a

        ⋮
```

- Type classes, functional dependencies

```
class Monad m a | a → m where
    return :: a → m a
    (>>=) :: a → (a → m b) → m b

        ⋮
```

More Background Work: Qualified Types

$$\Gamma \vdash M : \sigma$$

"Type of M is σ
and Γ specifies the free variables in M"

$$P \mid \Gamma \vdash M : \sigma$$

"Type of M is σ
when predicates in P are satisfied
and Γ specifies the free variables in M"[3]

Incorporate predicates into type language for finer grained polymorphism

More Background Work: Qualified Types

$$P \mid \Gamma \vdash M : \sigma$$

"Type of M is σ
when predicates in P are satisfied
and Γ specifies the free variables in M"[3]

Incorporate predicates into type language for finer grained polymorphism

$$(P \mid \sigma)$$

Instances of σ that satisfy P

More Background Work: Quill

Quill[7]: Qualified types + linear logic

Predicates:

- $\texttt{Un}\ \tau$ If τ does not have resources or can be copied or dropped easily.
- $\texttt{Fun}\ \tau$ If τ is a function type
- $\tau \geq \tau'$ If τ less restricting than τ'

More Background Work: Quill

Quill[7]: Qualified types + linear logic

Qualifying Types:

- Unrestricted Types: `Un Int`, `Un` Bool
- Restricted or Linear Types: FileHandle
- Function Types: `Fun` $(\texttt{Int} \rightarrow \texttt{Int})$, `Fun` $(\texttt{String} \rightarrow \texttt{String})$

QuB: Types and Predicates

$$
\begin{array}{rl}
\text{Types} & \tau, \upsilon, \phi ::= \mathbf{t} \mid \iota \mid \tau \to \tau \\
& \quad \text{where} \qquad \to \in \{\overset{!}{\ast\!\!\!\!\to}, \ast\!\!\!\!\to, \overset{!}{\twoheadrightarrow}, \twoheadrightarrow\} \\
\text{Predicates} & \pi, \omega ::= \mathtt{Un}\ \tau \mid \mathtt{ShFun}\ \phi \mid \mathtt{SeFun}\ \phi \mid \tau \geq \tau'
\end{array}
$$

- $\mathtt{SeFun}\ \phi$: ϕ is a function that is separate from its argument
- $\mathtt{ShFun}\ \phi$: ϕ is a function that is in sharing with its argument
- $\mathtt{Un}\ \tau$: τ does not have resources or they can be copied/dropped easily

QuB: Types and Predicates

$$\text{Types} \quad \tau, \upsilon, \phi ::= t \mid \iota \mid \tau \to \tau$$

$$\text{where} \qquad \to \in \{\twoheadrightarrow\!\!\!\!\!*, \twoheadrightarrow\}$$

$$\text{Predicates} \quad \pi, \omega ::= \texttt{Un}\ \tau \mid \texttt{ShFun}\ \phi \mid \texttt{SeFun}\ \phi \mid \tau \geq \tau'$$

- $\rightarrow\!\!\!\!\!*$: Function type that is separate from its argument
- $\twoheadrightarrow$: Function type that is in sharing with its argument
- $\overset{!}{\rightarrow\!\!\!\!\!*}$, $\overset{!}{\twoheadrightarrow}$: Unrestricted versions of $\rightarrow\!\!\!\!\!*$ and $\twoheadrightarrow$

- Kind System with type constructors and qualified types[1, 4]

$$
\begin{aligned}
& t, u \in \text{Type Variables} \\
\text{Kinds} \quad & \kappa ::= \star \mid \kappa' \to \kappa \\
\text{Types} \quad & \tau^{\kappa}, \phi^{\kappa} ::= t^{\kappa} \mid T^{\kappa} \mid \tau^{\kappa' \to \kappa} \tau^{\kappa'} \\
\text{Type Constructors} \quad & T^{\kappa} \in \mathcal{T}^{\kappa} \\
\text{where} \quad & \{\otimes, \&, \oplus, \overset{!}{\to\!\!\ast}, \to\!\!\ast, \overset{!}{\twoheadrightarrow}, \twoheadrightarrow\} \subseteq \mathcal{T}^{\star \to \star \to \star} \\
\text{Predicates} \quad & \pi, \omega ::= \mathtt{Un}\ \tau \mid \mathtt{SeFun}\ \phi \mid \mathtt{ShFun}\ \phi \mid \tau \geq \tau'
\end{aligned}
$$

QuB: Extension

- Sharing Pair

```
data ShPair a b = ShP a b

fst :: ShPair a b ↠ a
fst (ShP a b) = a              {- Succeeds typecheck -}

snd :: ShPair a b ↠ b
snd (ShP a b) = b              {- Succeeds typecheck -}
```

QuB: Extension

- Sharing Pair

```
data ShPair a b = ShP a b

fst :: ShPair a b ↠ a
fst (ShP a b) = a           {- Succeeds typecheck -}

snd :: ShPair a b ↠ b
snd (ShP a b) = b           {- Succeeds typecheck -}
```

- Separating Pair

```
data SePair a b = SeP a b

fst :: SePair a b -* a
fst (SeP a b) = a           {- Fails typecheck -}

swap :: SePair a b -* SePair b a
swap (SeP a b) = SePair b a {- Succeeds typecheck -}
```

What about filehandles, exceptions and memory leaks and runtime crashes?

Filehandles Revisited

File Handling API in QuB

$$\texttt{openFile} :: \texttt{FilePath} \multimap \texttt{IO FileHandle}$$

$$\texttt{closeFile} :: \texttt{FileHandle} \multimap \texttt{IO} \ ()$$

$$\texttt{readLine} :: \texttt{FileHandle} \multimap \texttt{IO} \ (\texttt{String}, \texttt{FileHandle})$$

$$\begin{aligned}\texttt{writeFile} :: \texttt{String} &\multimap \texttt{FileHandle}\\ &\multimap \texttt{IO} \ ((), \texttt{FileHandle})\end{aligned}$$

$$(\gg=) :: \texttt{IO a} \multimap (\texttt{a} \multimap \texttt{IO b}) \multimap \texttt{IO b}$$

Filehandles Revisited

```
do f  ← openFile "sample.txt"
   (s, f)  ← readLine f
   ()  ← closeFile f
   ()  ← closeFile f

(>>=) (openFile "sample.txt") (\ f →

(>>=) (readLine f) (\ (s, f) →

(>>=) (closeFile f) (\ _ → closeFile f)
```

Filehandles Revisited

```
do f   ← openFile "sample.txt"
   (s, f)  ← readLine f
   ()  ← closeFile f
   ()  ← closeFile f

(>>=) (openFile "sample.txt") (\ f →

(>>=) (readLine f) (\ (s, f) →

(>>=) (closeFile f) (\ _ → closeFile f)
```

Fails Typecheck!

```
do f  ← openFile "sample.txt"
   (s, f)  ← readLine f
   () ← closeFile f
   () ← closeFile f
```

```
{- (>>=) :: IO a ⊸ (a ⊸ IO b) ⊸ IO b -}
(>>=) (openFile "sample.txt") (\ f →
{- (>>=) :: IO a ⊸ (a ⊸ IO b) ⊸ IO b -}
(>>=) (readLine f) (\ (s, f) →
{- (>>=) :: IO a ⊸ (a ⊸ IO b) ↠ IO b -}
(>>=) (closeFile f) (\ _ → closeFile f)
```

Exceptions Revisited

```
openFile :: FilePath ⇾ IO FileHandle
closeFile :: FileHandle ⇾ IO ()
readFile :: FileHandle ⇾ IOF (String, FileHandle)
writeFile :: String ⇾ FileHandle ⇾ IOF ((), FileHandle)

throw :: Exception ⇾ IOF a
catch :: IOF a ⇾ (Exception ⇾ IO a) ↠ IO a
```

- May not fail `IO a`
- May fail `IOF a`

Exceptions Revisited

```
readFromFile :: FilePath ⇸ IO (Either String String)
readFromFile fpath =
do fh  ← openFile fpath
   ((s, fh)  ← readLine fh
   let l = caps s
   () ← closeFile fh
   return $ Right l) `catch`
           (\e → do closeFile fh
           return $ Left "Could not read file")
```

- Filehandle `fh` is shared between the `catch` arguments
- Avoids memory leak

Contributions revisited

- Design and implement QuB type system
    - Resources as first class citizens
    - Program objects are restricted or unrestricted
    - Functions that share resources with their arguments or are separate.
- Formalizing and proving important properties of QuB
- QuB is logic of ***BI*** on steroids
    - Typing Environments as graphs
- Working examples

Future Work

- Type inference algorithm $\mathcal{M}$ is incomplete. Terms can have two types.
    - $\{\texttt{Un}\ A\} \mid \varnothing \vdash \lambda^{\multimap} f.\lambda^{\multimap} x.fxx : (A \multimap A \multimap B) \multimap A \multimap B$
    - $\varnothing \mid \varnothing \vdash \lambda^{\multimap} f.\lambda^{\multimap} x.fxx : (A \multimap A \twoheadrightarrow B) \multimap A \twoheadrightarrow B$
- No formal semantic model yet

Thank You!

Q & A

References I

[1] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[2] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[3] Mark P. Jones. A theory of qualified types. *Science of Computer Programming*, 22(3):231 – 256, 1994.

[4] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*. Springer-Verlag LNCS 1782, 2000.

[5] Joachim Lambek. The mathematics of sentence structure. 65(3):154–170, 1958.

[6] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343. ACM, 1995.

References II

[7] J. Garrett Morris. The best of both worlds: Linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 448–461. ACM, 2016.

[8] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999.