QuB

A Resource Aware Functional Programming Language

Apoorv Ingle

The University of Kansas

Hard problems in programming

Naming variables

Hard problems in programming

Resource management

in evolving production code

Resources: Files, database connections, shared mutable state

Resource Management: File Handling

- Modified File Handling API in Haskell

$$
\begin{aligned}
\mathtt{openFile} &:: \mathtt{FilePath} \to \mathtt{IO\ FileHandle} \\
\mathtt{closeFile} &:: \mathtt{FileHandle} \to \mathtt{IO}\ () \\
\mathtt{readLine} &:: \mathtt{FileHandle} \to \mathtt{IO}\ (\mathtt{String}, \mathtt{FileHandle}) \\
\mathtt{writeFile} &:: \mathtt{String} \to \mathtt{FileHandle} \\
&\qquad \to \mathtt{IO\ FileHandle}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{upper} &:: \mathtt{String} \to \mathtt{String} \\
(\gg\!=) &:: \mathtt{IO\ FileHandle} \to (\mathtt{FileHandle} \to \mathtt{IO\ b}) \to \mathtt{IO\ b}
\end{aligned}
$$

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`[1] type class in Haskell

$$\begin{array}{l} \texttt{class Monad m} \Rightarrow \texttt{MonadError e m} \mid \texttt{m} \rightarrow \texttt{e where} \\ \quad \texttt{throwError} :: \texttt{e} \rightarrow \texttt{m a} \\ \quad \texttt{catchError} :: \texttt{m a} \rightarrow (\texttt{e} \rightarrow \texttt{m a}) \rightarrow \texttt{m a} \end{array}$$

- `MonadError` instance with `IO` and `Exception`

$$\begin{array}{l} \texttt{throwError} :: \texttt{Exception} \rightarrow \texttt{IO a} \\ \texttt{catchError} :: \texttt{IO a} \rightarrow (\texttt{Exception} \rightarrow \texttt{IO a}) \rightarrow \texttt{IO a} \end{array}$$

- `throwError` start exception processing
- `catchError` exception handler

[1] (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. New York, NY, USA: ACM, 1995, pp. 333–343)

Resource Management: Exception Handling

- Using `MonadError` in Haskell

```
(do f ← openFile "sample.txt"
   (s, f) ← readLine f        {- Exception raised here -}
    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 revisited

- Language design
  - Resources are "first class citizens"
  - Resources(variables) can be in sharing or separated
- QuB is logic of ***BI*** on steroids
  - Typing Environments as graphs
- Formalizing and proving important properties of QuB
  - Type system
  - Syntax directed type system (sound and complete)
  - Type inference algorithm $\mathcal{M}$ (sound)
- Working examples

Background Work: STLC and *HM* type system

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

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

Background Work: STLC and *HM* type system

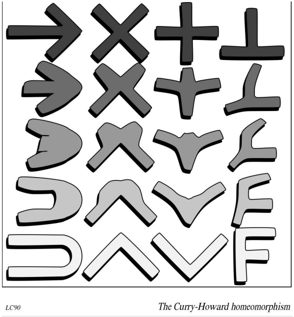
$$\lambda x.M \begin{cases} \text{Abstract over computation} \\ \text{Define functions} \end{cases} \qquad \lambda x.M : \tau \to \tau' \begin{cases} x : \tau \\ M : \tau' \end{cases}$$

$$MN \begin{cases} \text{Do the computation} \\ \text{Use functions} \end{cases} \qquad MN : \tau' \begin{cases} M : \tau \to \tau' \\ N : \tau \end{cases}$$

Background Work: Curry-Howard Correspondence

- 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

$\boxed{\text{Language}}$

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

$\boxed{\text{Implicit Structural Rules}}$

$$\begin{array}{cc} A, B \vdash A \quad A, B \vdash B & \text{Contraction} \\ A \vdash A \wedge A & \text{Weakening} \\ A, B \vdash B, A & \text{Exchange} \end{array}$$

Background Work: Second Order Intuitionistic Propositional Logic

Propositions are truth values not resources

$\boxed{\text{Language}}$

$$\text{Propostions \& connectives} \quad A, B, C ::= x \mid A \supset B \mid \forall x.B \mid A \vee B \mid A \wedge B$$

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

$\boxed{\text{Implicit Structural Rules}}$

$$\begin{array}{cl} A, B \vdash A \quad A, B \vdash B & \text{Contraction} \\ A \vdash A \wedge A & \text{Weakening} \\ A, B \vdash B, A & \text{Exchange} \end{array}$$

Background Work: Substructural Logic

System	Who	Year	Control
Revelance Logic[2]	Orlev	1928	[WKN]
Lambek Logic[3]	Lambek	1958	[EXCH]
Affine Logic[4]	Grishin	1974	[CTR]
Linear Logic[5]	Girard	1987	[WKN] [CTR]
Logic of Bunched Implications[6]	O'Hearn and Pym	1999	[WKN] [CTR]
Separation Logic [7]	Reynolds	2002	[WKN] [CTR]
⋮	⋮	⋮	⋮

[2] Ivan Orlov. 'The Logic of Compatibility of Propositions'. In: *Matematicheskii Sbornik* (1928)

[3] Joachim Lambek. 'The Mathematics of Sentence Structure'. In: *The American Mathematical Monthly* 65.3 (1958), pp. 154–170

[4] V Grishin. 'A nonstandard logic and its application to set theory'. Russian. In: *Studies in Formalized Languages and Nonclassical Logics* (1974)

[5] Jean-Yves Girard. 'Linear logic'. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101

[6] Peter W. O'Hearn and David J. Pym. 'The Logic of Bunched Implications'. In: *The Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244

[7] John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. 2002

Background work: Logic of ***BI***

Coffee Shop

1 cup coffee costs \$2

,

$\vdash$

Coffee Shop

1 cup coffee costs \$2

,

$\vdash$

,

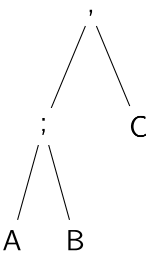
$\nvdash$

two separate dollar bills necessary

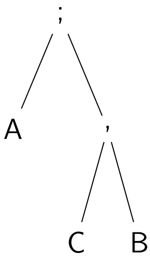
- Two meanings of conjunction:

  Separate or Shared

- 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

$$A, B \vdash A \otimes B \qquad\qquad A; B \vdash A \mathbin{\&} B$$

- Meaning of implication

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

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

QuB: Types

$$\text{Types} \quad \tau, \upsilon, \phi ::= \boldsymbol{t} \mid \iota \mid \tau \to \tau$$
$$\text{where} \qquad \to \in \{\twoheadrightarrow^{*}, \twoheadrightarrow\}$$

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

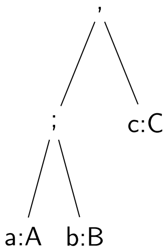
QuB: Expression Language

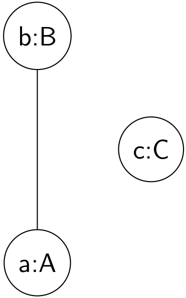
$$\begin{aligned} \text{Term Variables} \quad & x, y, z \in \mathrm{Var} \\ \text{Expressions} \quad & M, N ::= x \\ & \quad \mid \lambda^{-\!*} x.M \mid \lambda^{\twoheadrightarrow} x.M \\ & \quad \mid MN \mid \mathtt{let}\ x = M\ \mathtt{in}\ N \end{aligned}$$

- $\lambda^{-\!*} x.M$: 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



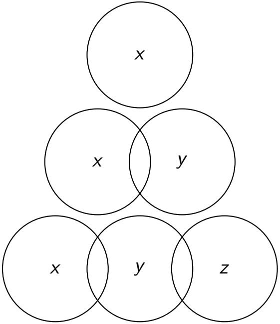


- Nodes are program objects
- (No) Edges represent (no) sharing

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)

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

- Additive Product (Sharing Pair)

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

- Sums

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

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

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

$$\begin{aligned}&\mathtt{data\ Bool\ \ \ } = \mathtt{True} \mid \mathtt{False}\\&\mathtt{data\ List\ a} = \mathtt{Nil} \mid \mathtt{Cons\ a\ a}\\&\mathtt{data\ Tree\ a} = \mathtt{Leaf} \mid \mathtt{Node\ a\ a}\end{aligned}$$

$$\vdots$$

- Type classes, functional dependencies

$$\begin{aligned}&\mathtt{class\ Monad\ m\ a} \mid \mathtt{a} \rightarrow \mathtt{m\ where}\\&\quad \mathtt{return} :: \mathtt{a} \rightarrow \mathtt{m\ a}\\&\quad (\gg\!=) :: \mathtt{a} \rightarrow (\mathtt{a} \rightarrow \mathtt{m\ b}) \rightarrow \mathtt{m\ b}\end{aligned}$$

$$\vdots$$

More Background Work: Qualified Types

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

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

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

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

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"[10]

Incorporate predicates into type language for finer grained polymorphism

$$(P \mid \sigma)$$

Instances of σ that satisfy P

More Background Work: Quill

Quill[11]: Qualified types + linear logic

Predicates:

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

More Background Work: Quill

Quill[12]: Qualified types + linear logic

Qualifying Types:

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

QuB: Types and Predicates

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

- $\mathtt{SeFun}\ \phi$: 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

$$
\begin{array}{lrl}
\text{Types} & \tau, \upsilon, \phi ::= & \boldsymbol{t} \mid \iota \mid \tau \to \tau \\
& \text{where} & \to \in \{-\!\!*, \twoheadrightarrow\} \\
\text{Predicates} & \pi, \omega ::= & \mathtt{Un}\ \tau \mid \mathtt{ShFun}\ \phi \mid \mathtt{SeFun}\ \phi \mid \tau \geq \tau'
\end{array}
$$

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

QuB: Extension

- Kind System with type constructors and qualified types[10, 11]

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

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 -}
```

Motivation Revisited

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)
```

Filehandles Revisited

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

- Language design
    - Resources are "first class citizens"
    - Resources(variables) can be in sharing or separated
- QuB is logic of ***BI*** on steroids
    - Typing Environments as graphs
- Formalizing and proving important properties of QuB
    - Type system
    - Syntax directed type system (sound and complete)
    - Type inference algorithm $\mathcal{M}$ (sound)
- Working examples

Future Work

- Type inference algorithm $\mathcal{M}$ is incomplete. Terms can have two types.
  - $\{\mathtt{Un}\ A\} \mid \varnothing \vdash \lambda^{\rightarrow\!\!*} f.\lambda^{\rightarrow\!\!*} x.fxx : (A \mathrel{\rightarrow\!\!*} A \mathrel{\rightarrow\!\!*} B) \mathrel{\rightarrow\!\!*} A \mathrel{\rightarrow\!\!*} B$
  - $\varnothing \mid \varnothing \vdash \lambda^{\rightarrow\!\!*} f.\lambda^{\rightarrow\!\!*} x.fxx : (A \mathrel{\rightarrow\!\!*} A \twoheadrightarrow B) \mathrel{\rightarrow\!\!*} A \twoheadrightarrow B$
- Current semantics: call-by-value.

  Formalize resource correctness.

Thank You!

Q & A

References I

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. New York, NY, USA: ACM, 1995, pp. 333–343.

Ivan Orlov. 'The Logic of Compatibility of Propositions'. In: *Matematicheskii Sbornik* (1928).

Joachim Lambek. 'The Mathematics of Sentence Structure'. In: *The American Mathematical Monthly* 65.3 (1958), pp. 154–170.

V Grishin. 'A nonstandard logic and its application to set theory'. Russian. In: *Studies in Formalized Languages and Nonclassical Logics* (1974).

Jean-Yves Girard. 'Linear logic'. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101.

References II

Peter W. O'Hearn and David J. Pym. 'The Logic of Bunched Implications'. In: *The Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244.

John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. 2002.

Mark P. Jones. 'A theory of qualified types'. In: *Science of Computer Programming* 22.3 (1994), pp. 231 –256.

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. New York, NY, USA: ACM, 2016, pp. 448–461.

Henk Barendregt. 'Introduction to generalized type systems'. In: *Journal of Functional Programming* 1.2 (1991), 125–154.

References III

Mark P. Jones. 'Type Classes with Functional Dependencies'. In: *Proceedings of the 9th European Symposium on Programming*. ESOP 2000. Berlin, Germany: Springer-Verlag LNCS 1782, 2000.