The Curry-Howard Correspondence: Bridging Types and Logics

Major Area Exam

Candidate: Miroslav Gavrilov

Ben Hardekopf Chandra Krintz Tevfik Bultan

Outline

Motivation

- **Intuitionistic logic** and **simply-typed lambda calculus**
- **Classical logic** and **lambda-mu calculus**

Several interesting findings

A more refined perspective

Conclusion

Outline

Motivation

Several interesting findings

- **Classical linear logic** and **process calculi**
- **Predicate logic** and **information-flow types**

A more refined perspective

Conclusion

Outline

Motivation

Several interesting findings

A more refined perspective

Functors and **general refinement systems**

Conclusion

Outline

Motivation

Several interesting findings

A more refined perspective

Conclusion

- **Review**
- **Opportunities**

Outline

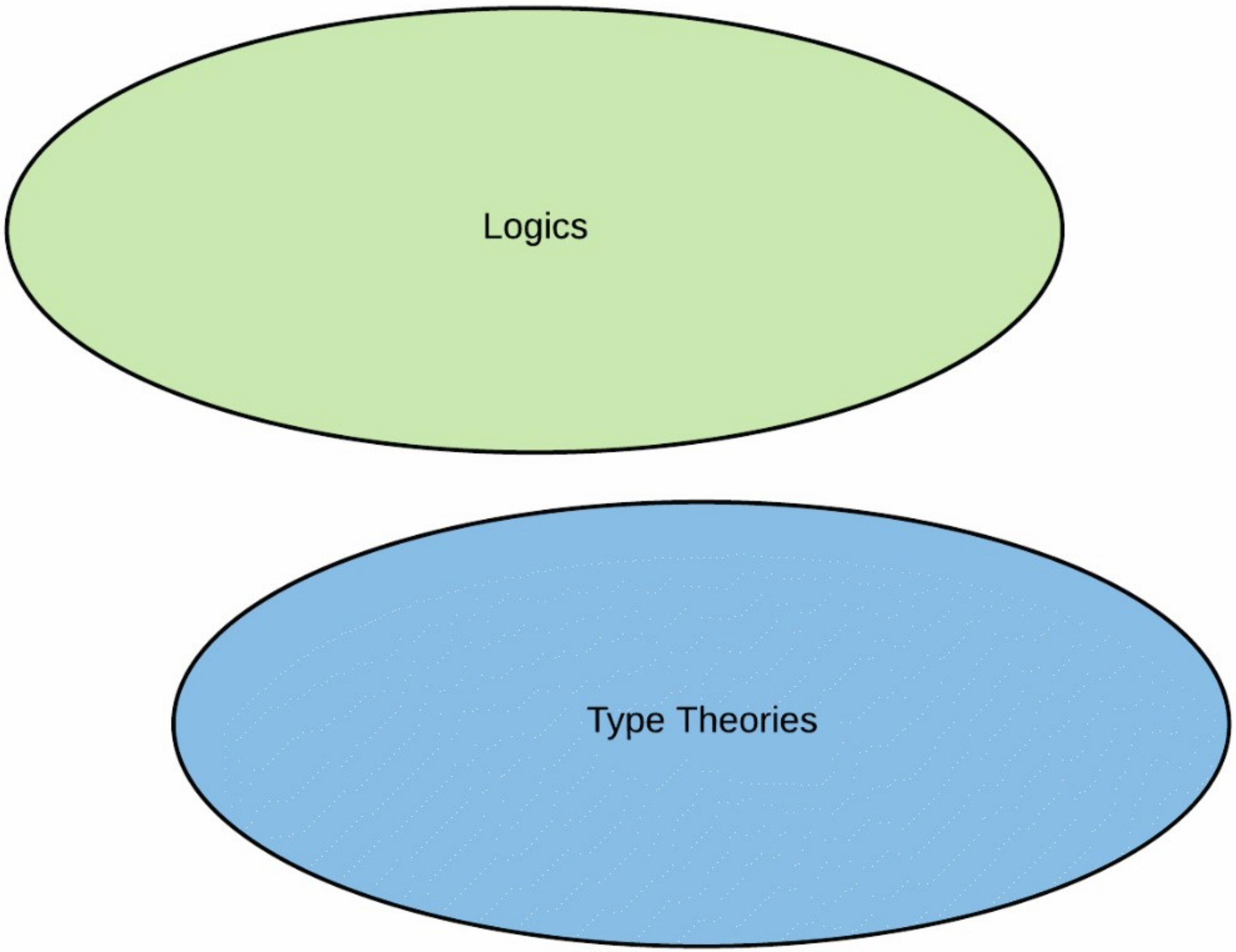
Motivation

Several interesting findings

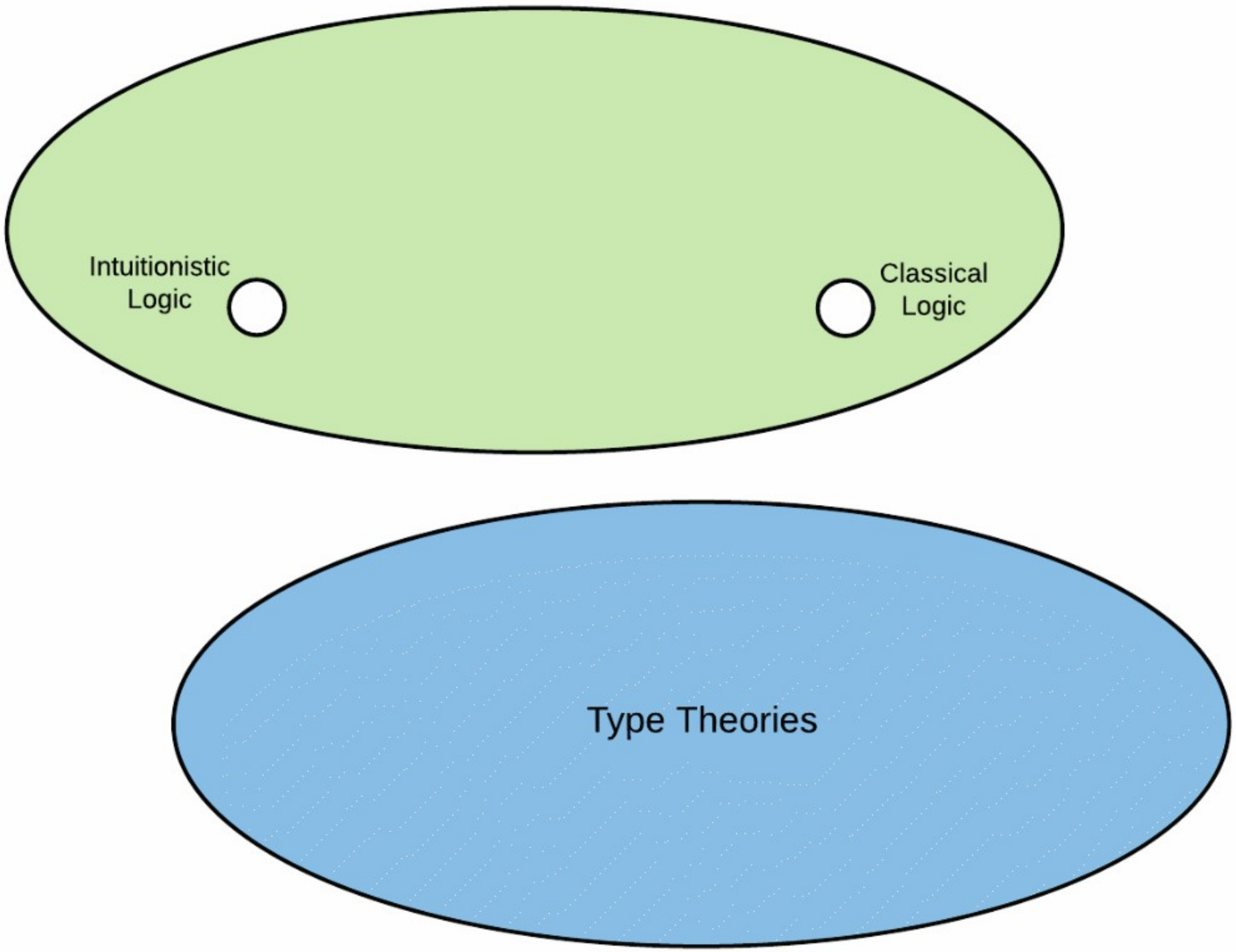
A more refined perspective

Conclusion

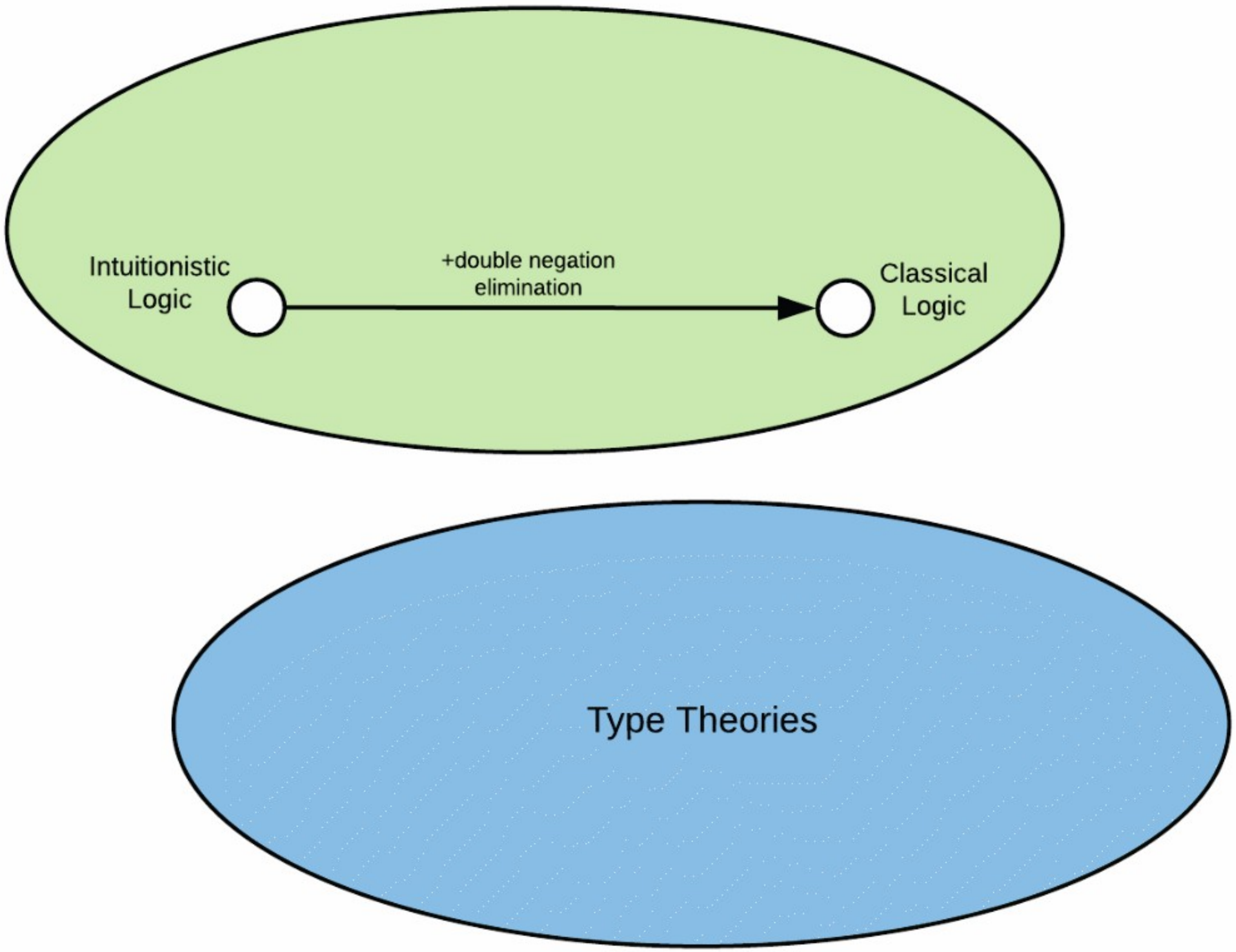
Motivation



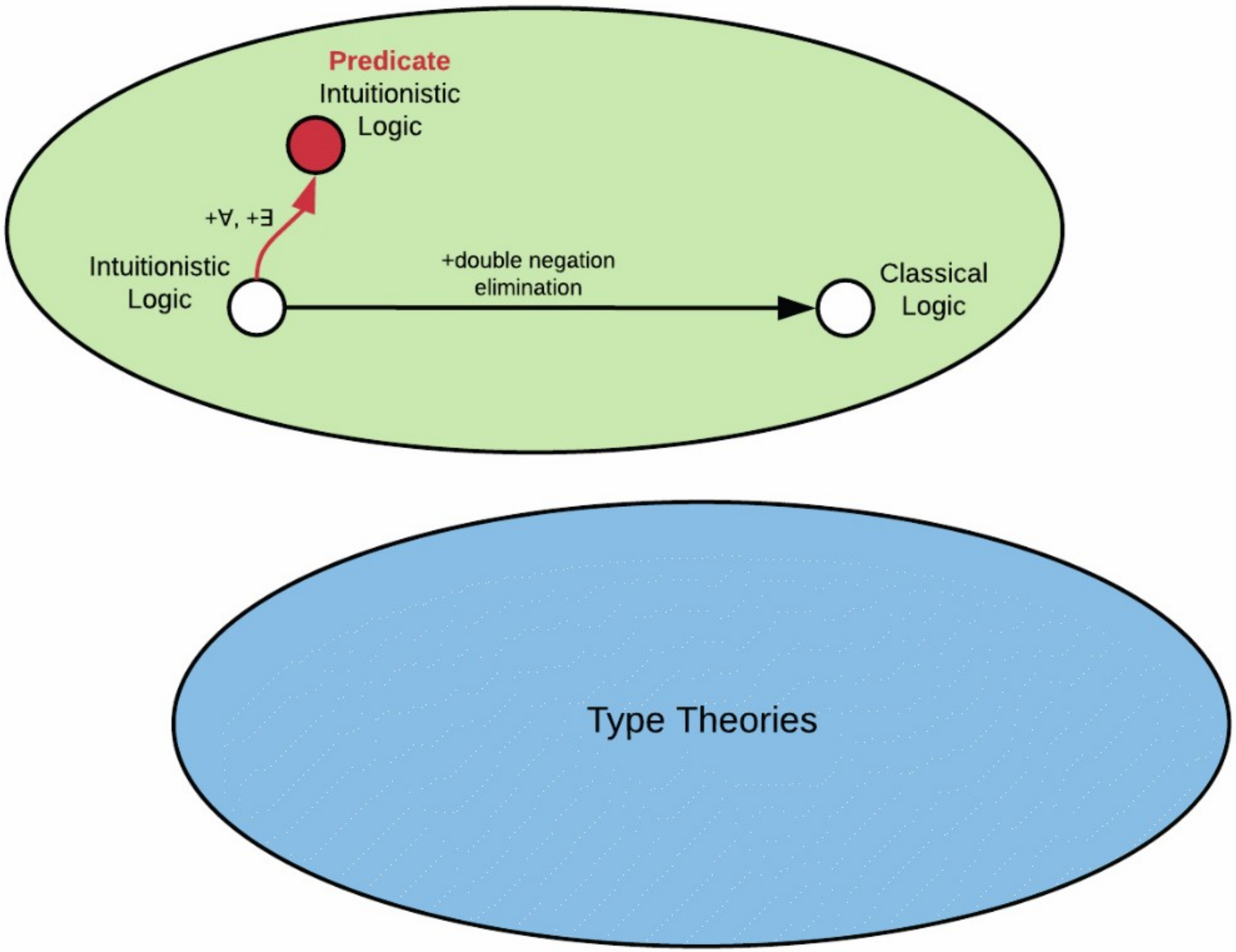
Motivation



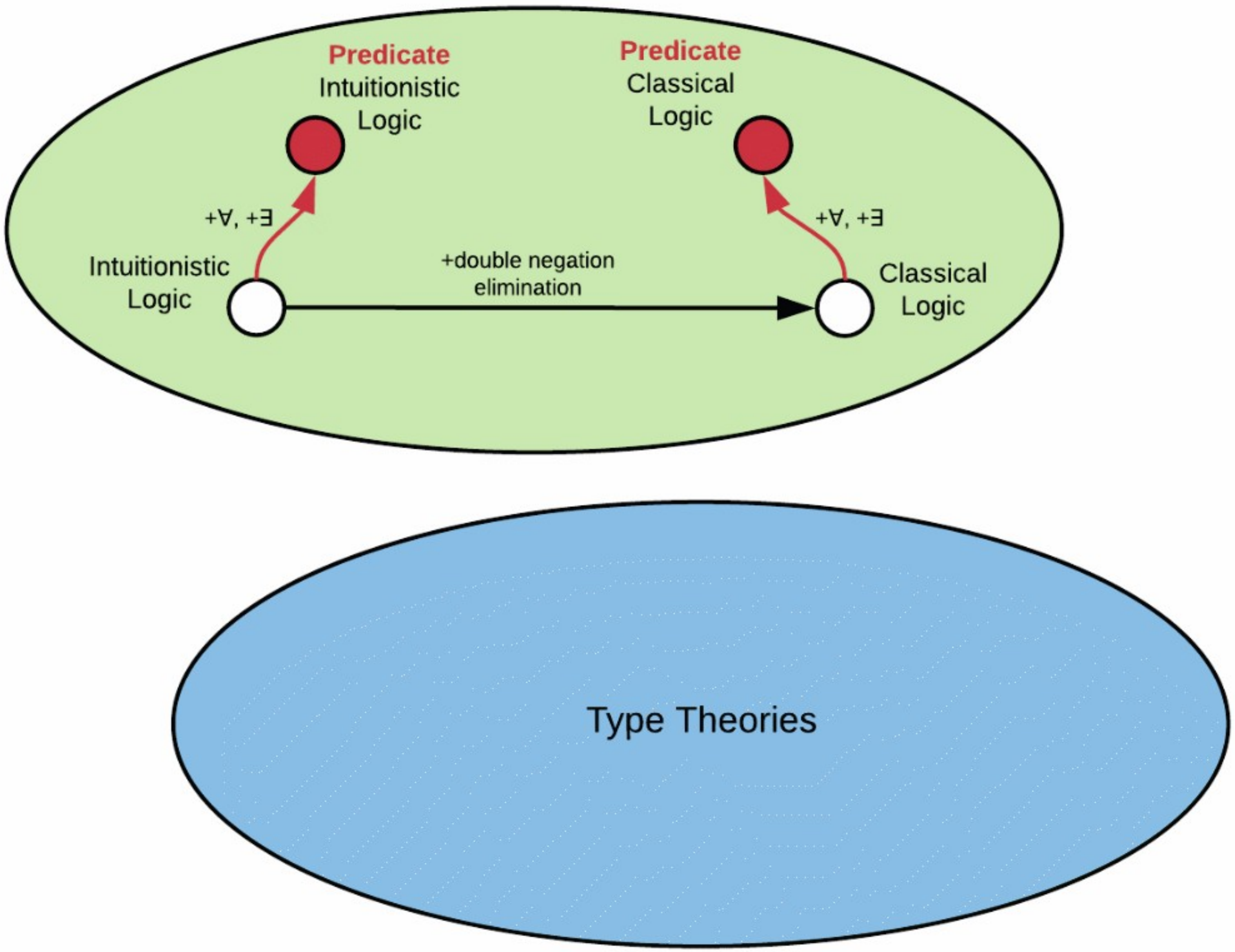
Motivation


Intuitionistic Logic
+double negation elimination
Classical Logic
Type Theories

Motivation

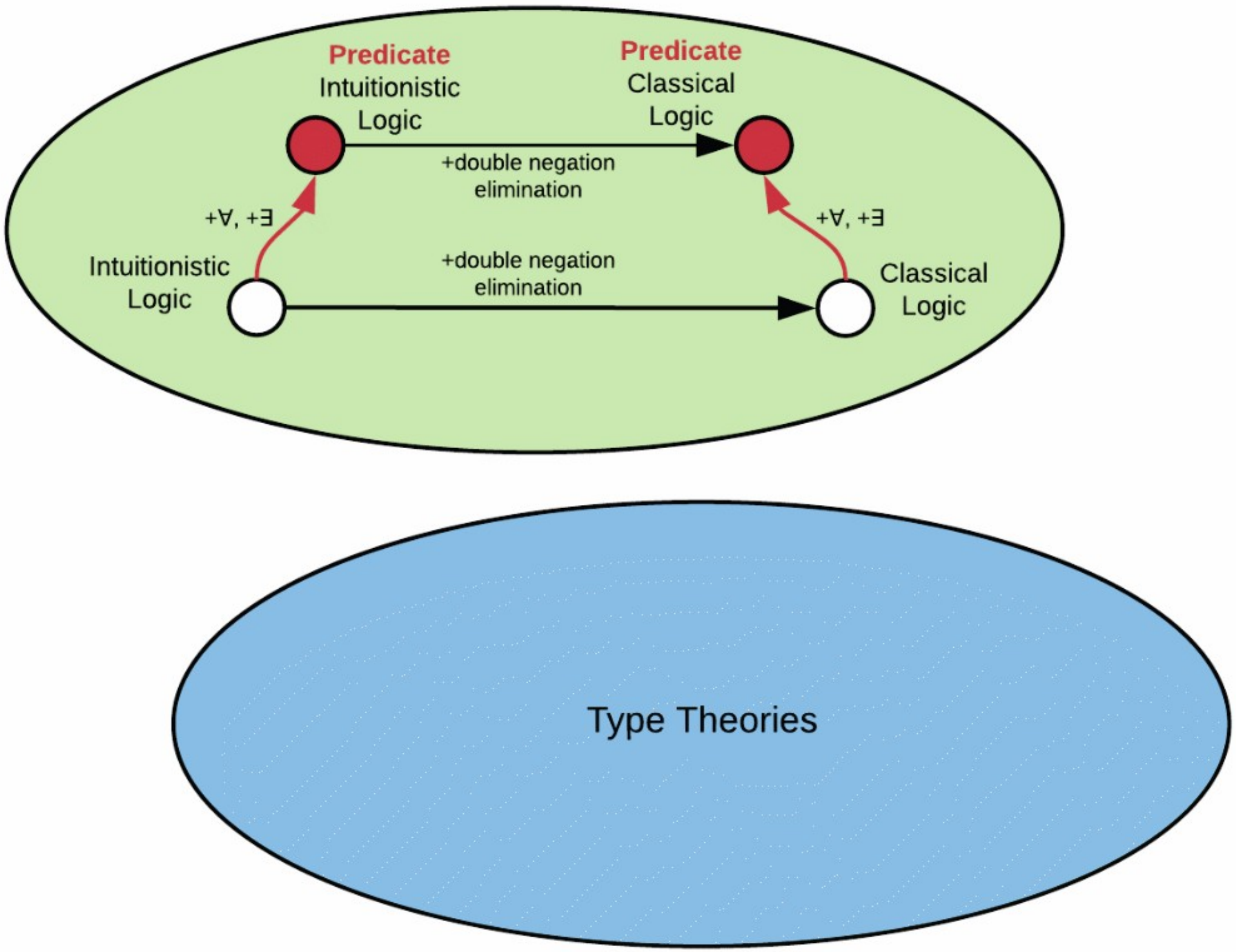

Predicate
Intuitionistic
Logic
+∀, +∃
Intuitionistic
Logic
+double negation
elimination
Classical
Logic
Type Theories

Motivation



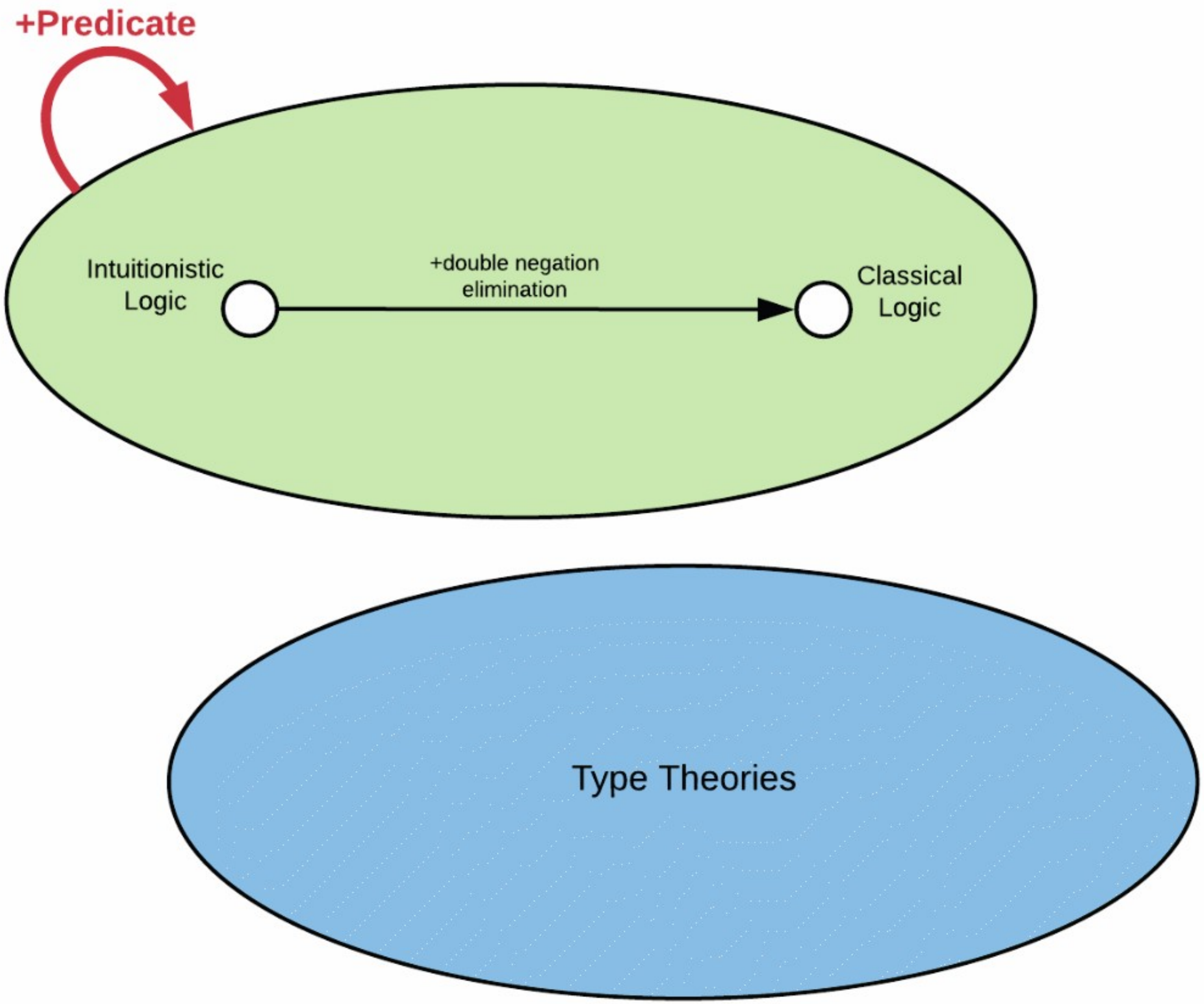
Predicate
Intuitionistic
Logic
Predicate
Classical
Logic
+∀, +∃
+∀, +∃
Intuitionistic
Logic
+double negation
elimination
Classical
Logic
Type Theories

Motivation



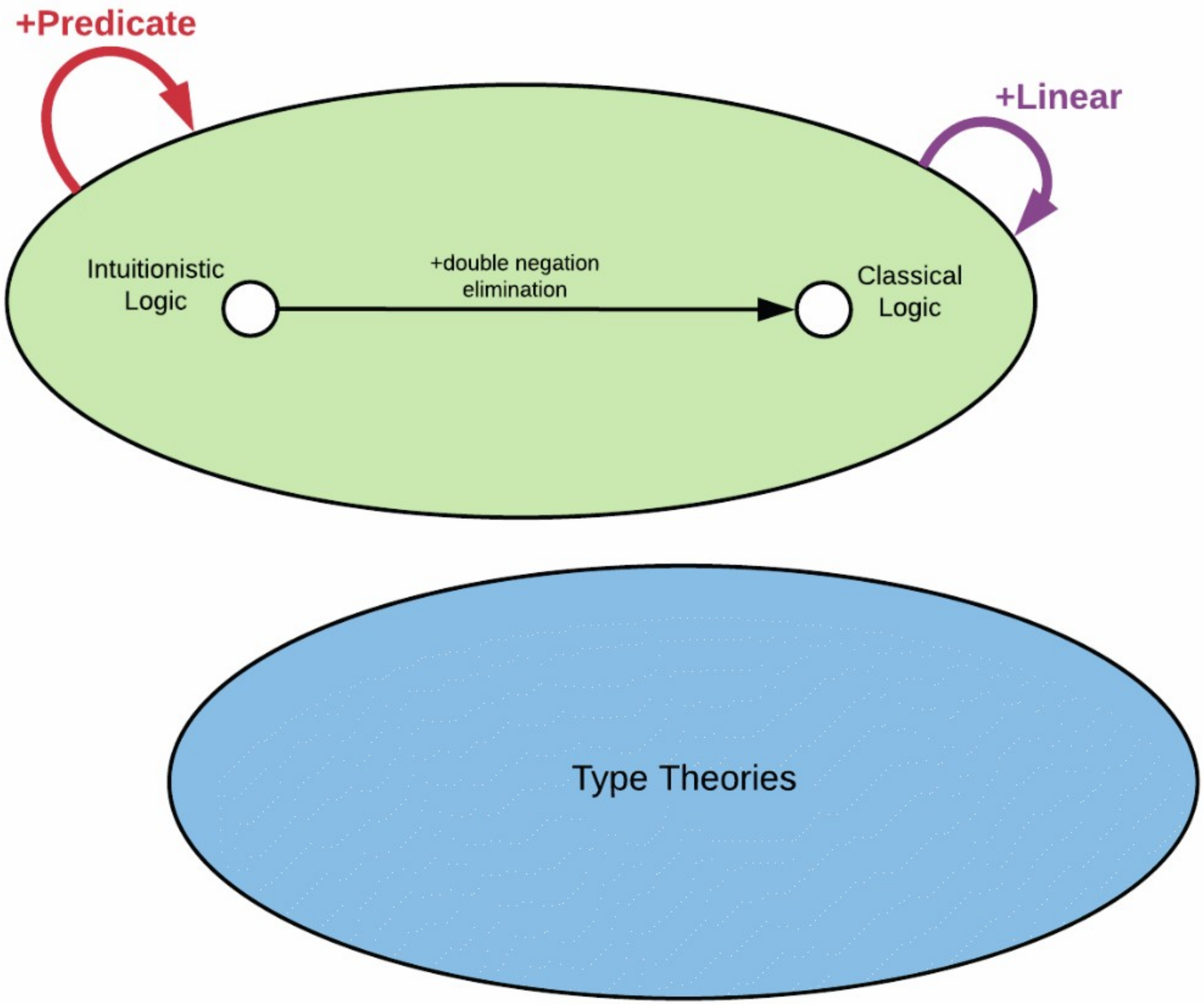
Predicate
Intuitionistic
Logic
Predicate
Classical
Logic
+double negation
elimination
+∀, +∃
+∀, +∃
Intuitionistic
Logic
+double negation
elimination
Classical
Logic
Type Theories

Motivation



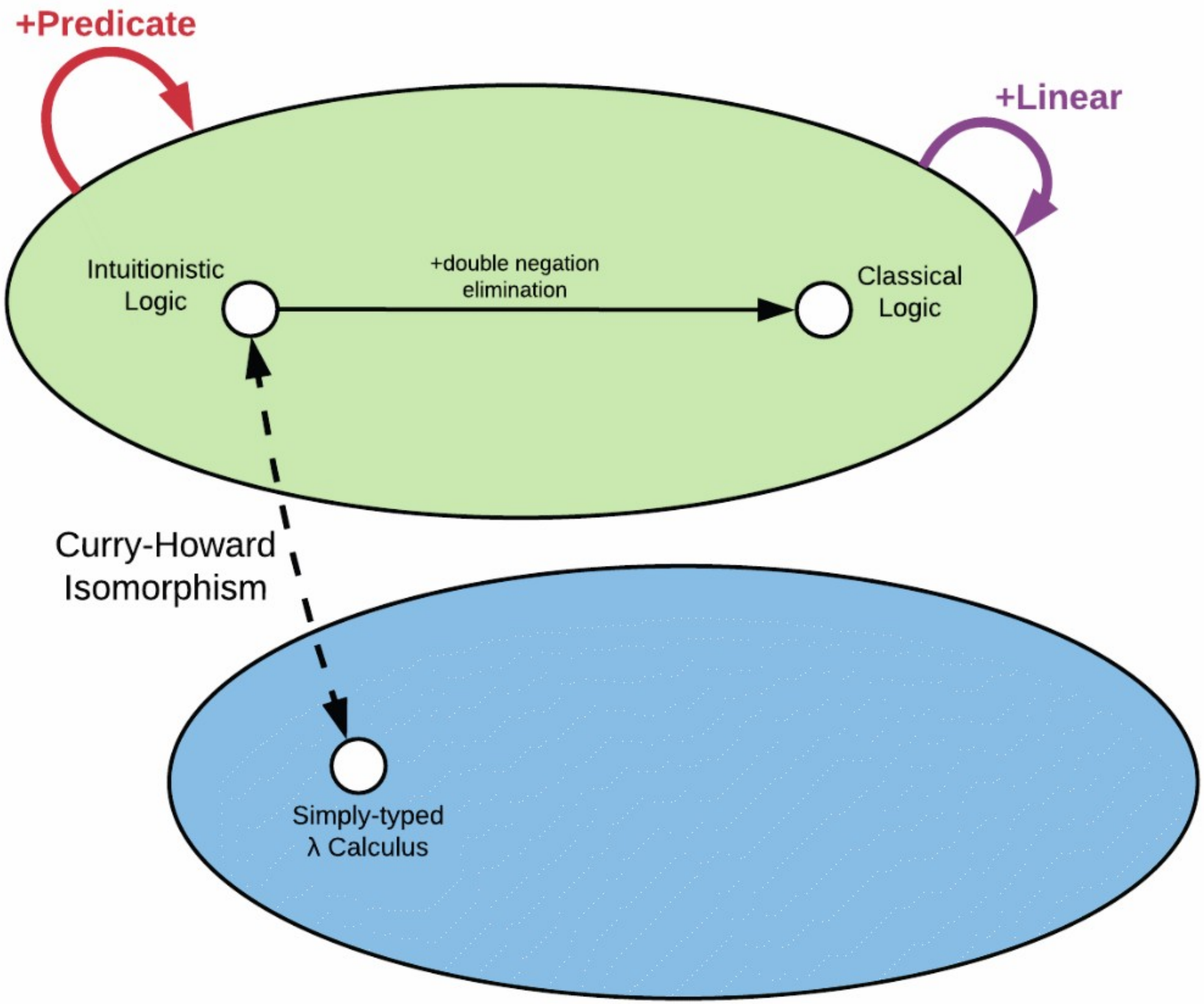
+Predicate
Intuitionistic Logic
+double negation elimination
Classical Logic
Type Theories

Motivation

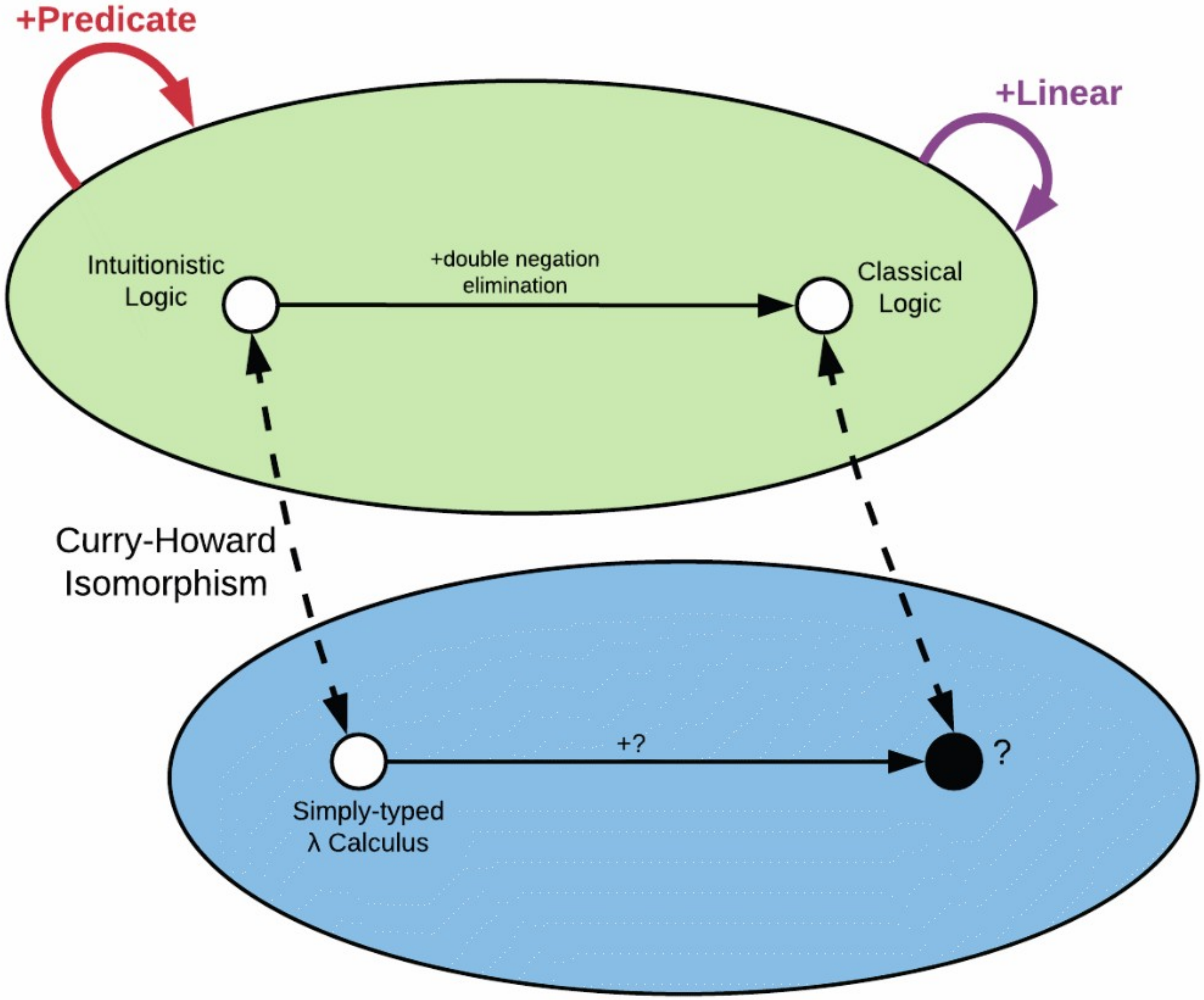


+Predicate
+Linear
Intuitionistic Logic
+double negation elimination
Classical Logic
Type Theories

Motivation

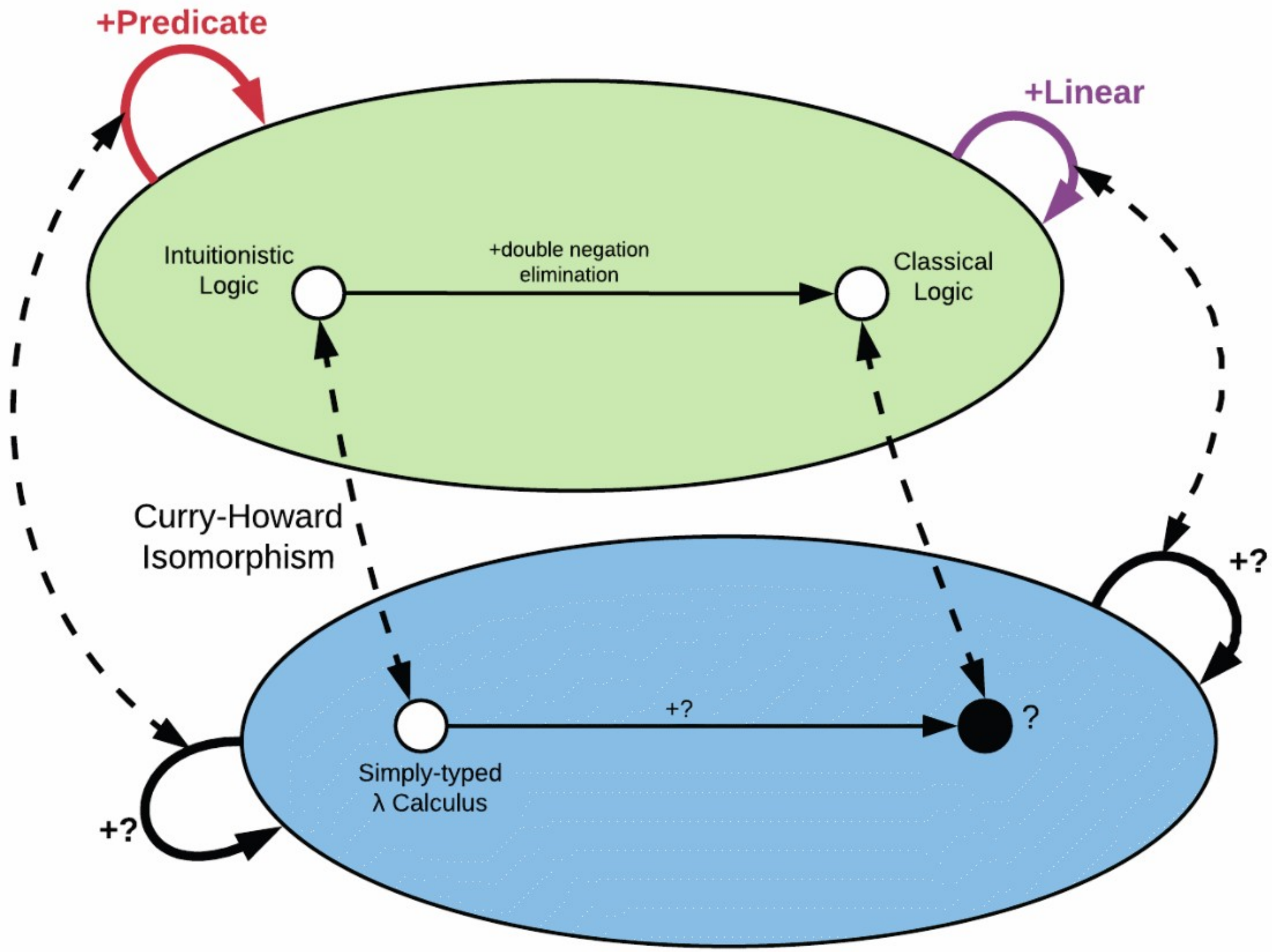

+Predicate
+Linear
Intuitionistic
Logic
+double negation
elimination
Classical
Logic
Curry-Howard
Isomorphism
Simply-typed
λ Calculus

Motivation



+Predicate
+Linear
Intuitionistic Logic
+double negation elimination
Classical Logic
Curry-Howard Isomorphism
+?
?
Simply-typed λ Calculus

Motivation



+Predicate
+Linear
Intuitionistic Logic
+double negation elimination
Classical Logic
Curry-Howard Isomorphism
+?
+?
Simply-typed λ Calculus
+?
?
+?

Motivation

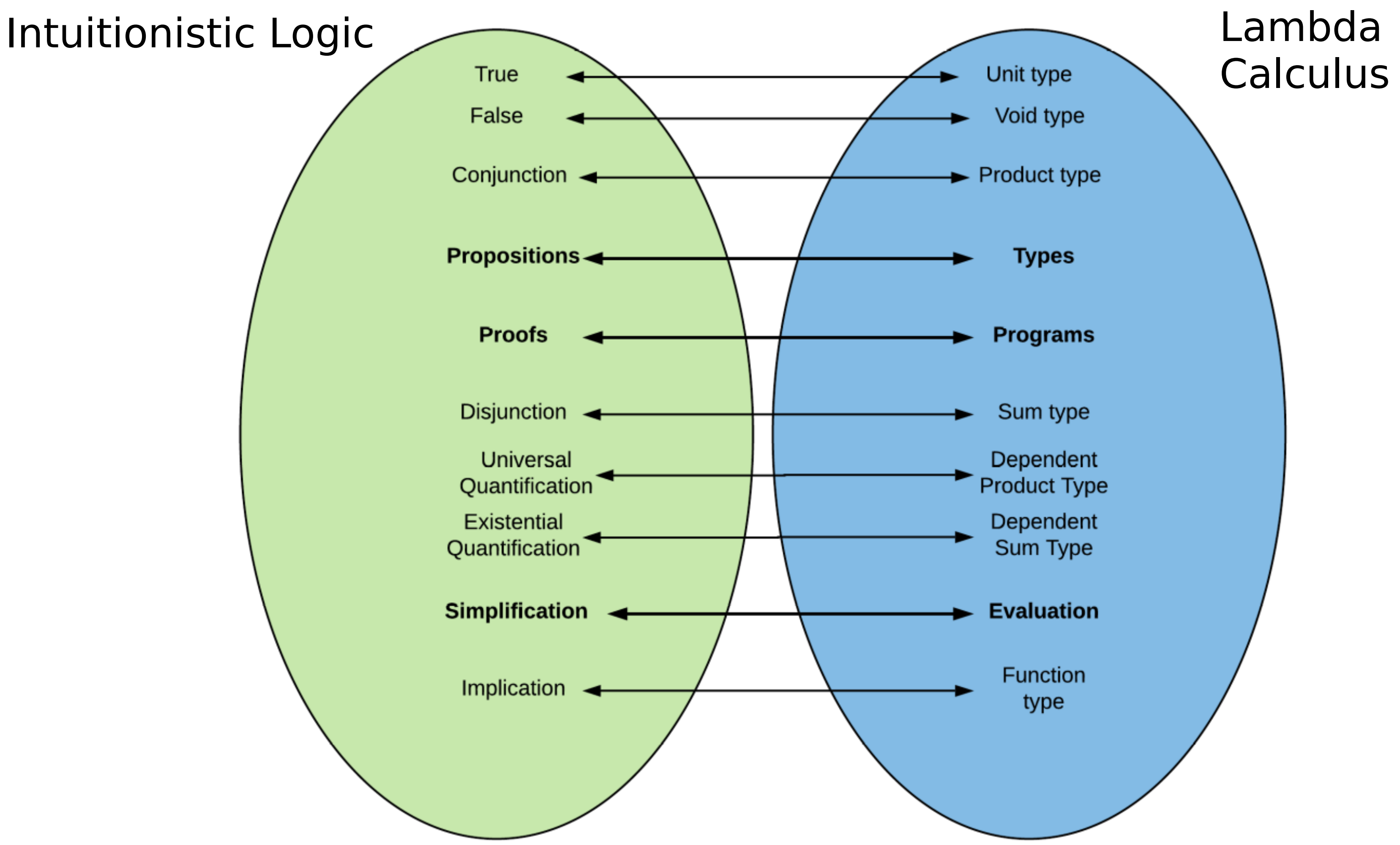
Haskell Curry

Logic	equals[1,2]	**Types**
$A \Rightarrow B$	$\equiv$	$A \rightarrow B$
$A \vee B$	$\equiv$	$A + B$
$A \wedge B$	$\equiv$	$A \times B$

William Alvin Howard

[1] H. B. Curry. Functionality in combinatory logic. Proceedings of the National Academy of Science, 20:584–590, 1934.
[2] W. A. Howard. The formulae-as-types notion of construction. In To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pages 479–491. Academic Press, 1980.

Motivation



Intuitionistic Logic
Lambda Calculus
True
False
Conjunction
Propositions
Proofs
Disjunction
Universal Quantification
Existential Quantification
Simplification
Implication
Unit type
Void type
Product type
Types
Programs
Sum type
Dependent Product Type
Dependent Sum Type
Evaluation
Function type

Lambda Calculus

Simply-typed Lambda Calculus

$$e \ ::= \ x \ \mid \ \lambda\, x \ : \ \tau \ . \ e \ \mid \ e_1\ e_2$$

$$\tau \ ::= \ \tau_1 \rightarrow \tau_2 \ \mid \ B$$

Lambda Calculus

Simply-typed Lambda Calculus Reduction

$$\frac{\mathbf{sqrd} \equiv \lambda\, x : \mathbb{N} \,.\, \mathbf{mult}\; x\; x}{\begin{aligned} &\mathbf{sqrd}\; 4 \\ &\quad \equiv (\lambda\, x : \mathbb{N} \,.\, \mathbf{mult}\; x\; x)\; 4 \\ &\quad =_{\beta} \mathbf{mult}\; 4\; 4 \\ &\quad =_{\beta *} 16 \end{aligned}}$$

Lambda Calculus

Simply-typed Lambda Calculus Typing Rules: **App**

$$\frac{\Gamma \vdash x : \tau \to \sigma \quad \Gamma \vdash y : \tau}{\Gamma \vdash x\ y : \sigma} (3)$$

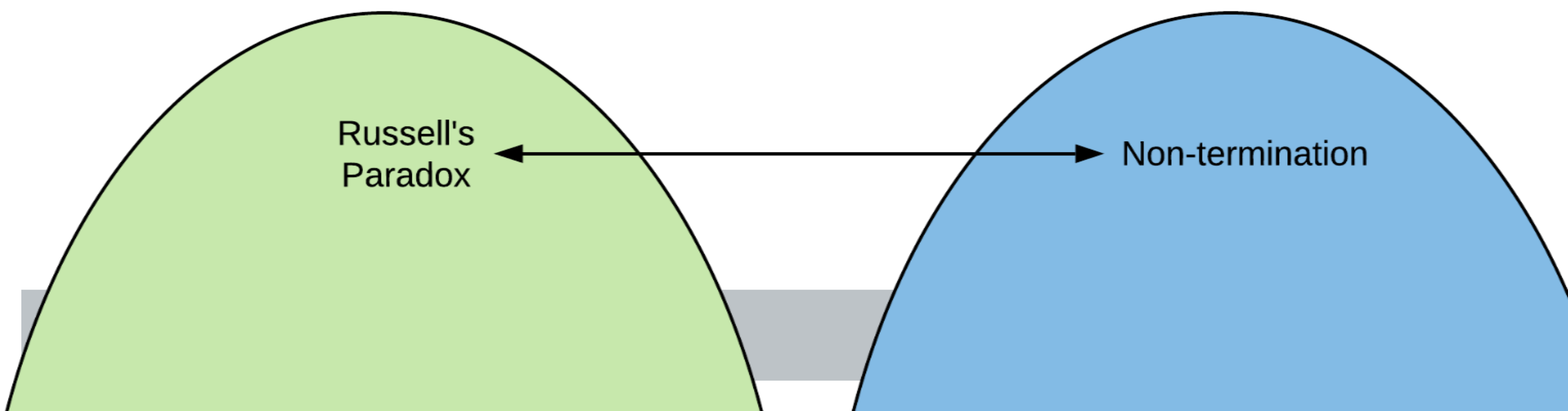
Modus Ponens: A ⇒ B, A ⊢ B

Lambda Calculus

Simply-typed Lambda Calculus Typing Rules: **App**

Most Typed Lambda Calculi are **strongly normalizing**:
every term can be written in a *normal form*, such that it doesn't reduce further

Strong normalization = All computations terminate



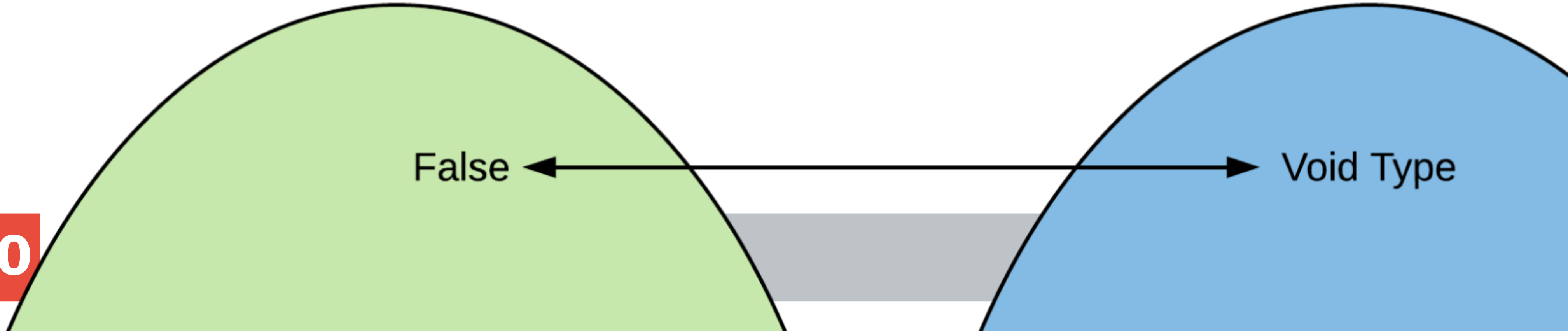
Curry-Howard Isomorphism

this object is **proof** that type **τ → τ** isn't empty

$$\lambda x : \tau . x$$

$$\tau \to \tau$$

$$T \Rightarrow T$$



Curry-Howard Isomorphism

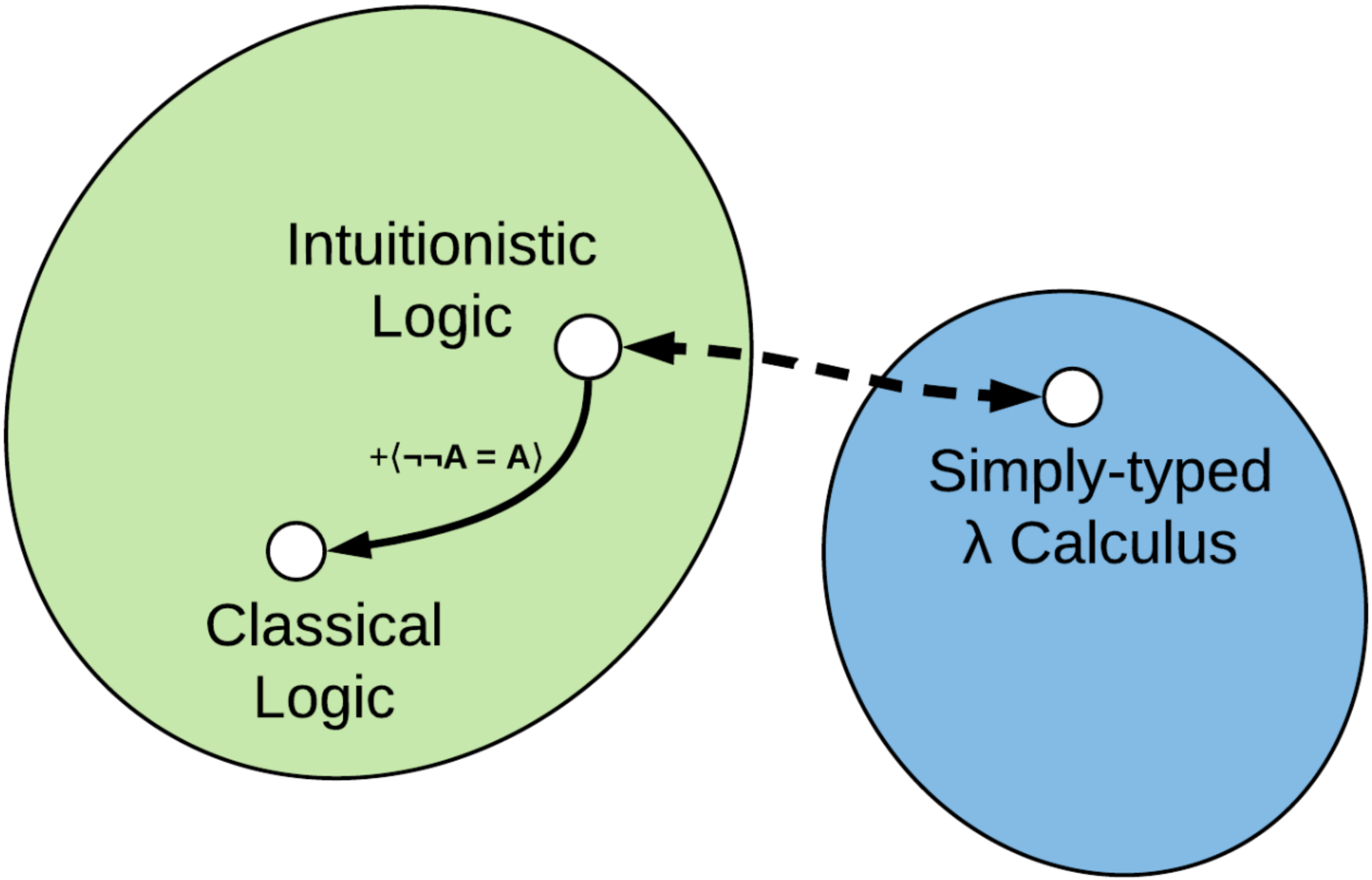
$$\neg A \equiv A \Rightarrow \bot$$

$$\neg\neg A \equiv (A \Rightarrow \bot) \Rightarrow \bot$$

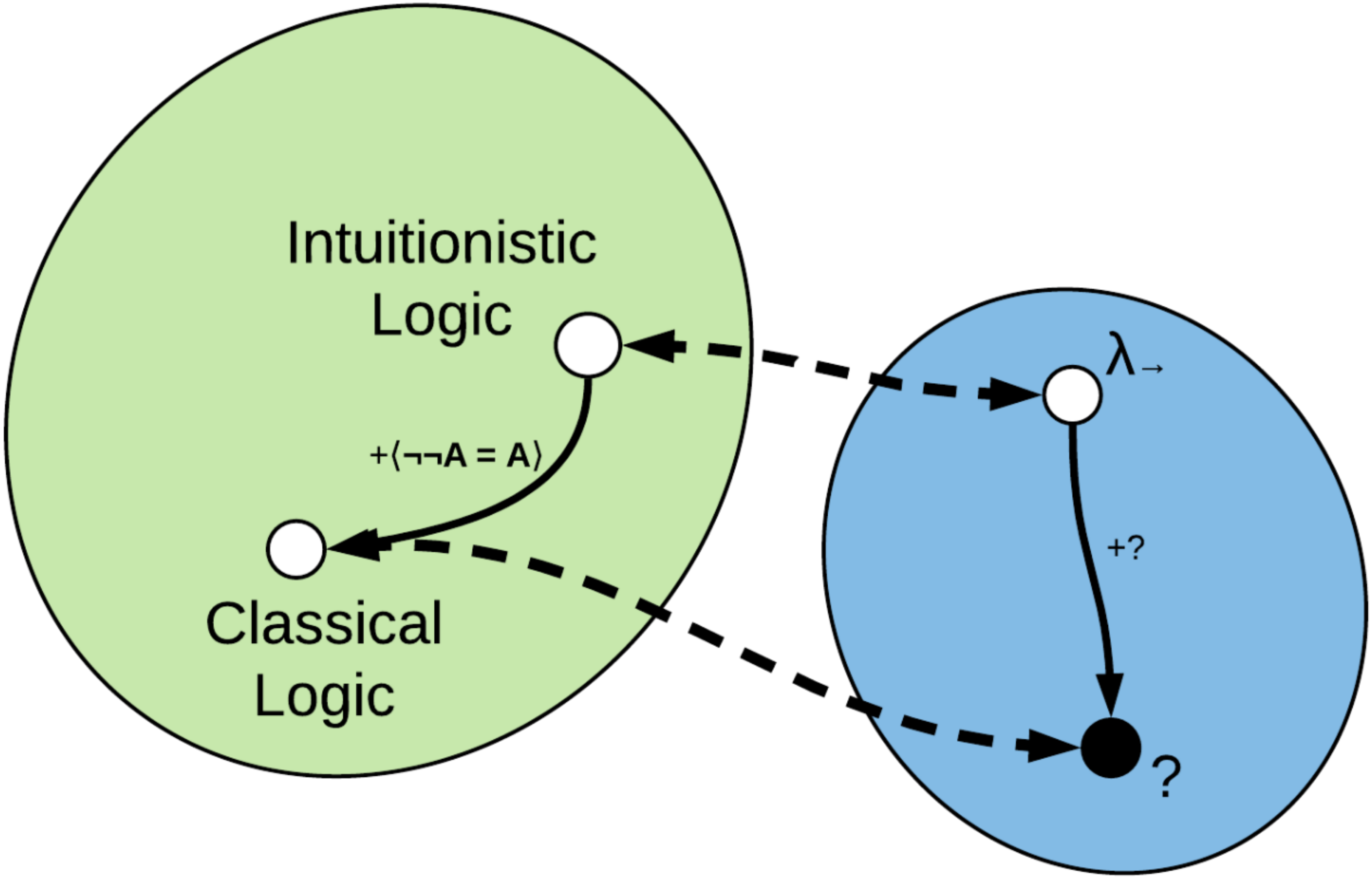
$$\cancel{(A \to \mathbf{Void}) \to \mathbf{Void}}$$

~~“If you give me a function that can turn **A** to **Void**, I can give you **Void**”~~

Curry-Howard Isomorphism



An open invitation



Extending the Correspondence

Parigot's λμ (lambda-mu) Calculus[3]

Introduces **named** terms (mu terms)

$$[\alpha]t$$

Control over **which parts** of an expression are reduced (commands)

$$\mu\ \alpha\ .\ E$$

[3] Michel Parigot. 1992. Lambda-My-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR '92), Andrei Voronkov (Ed.). Springer-Verlag, London, UK, UK, 190-201.

Extending the Correspondence

Parigot's λμ (lambda-mu) Calculus

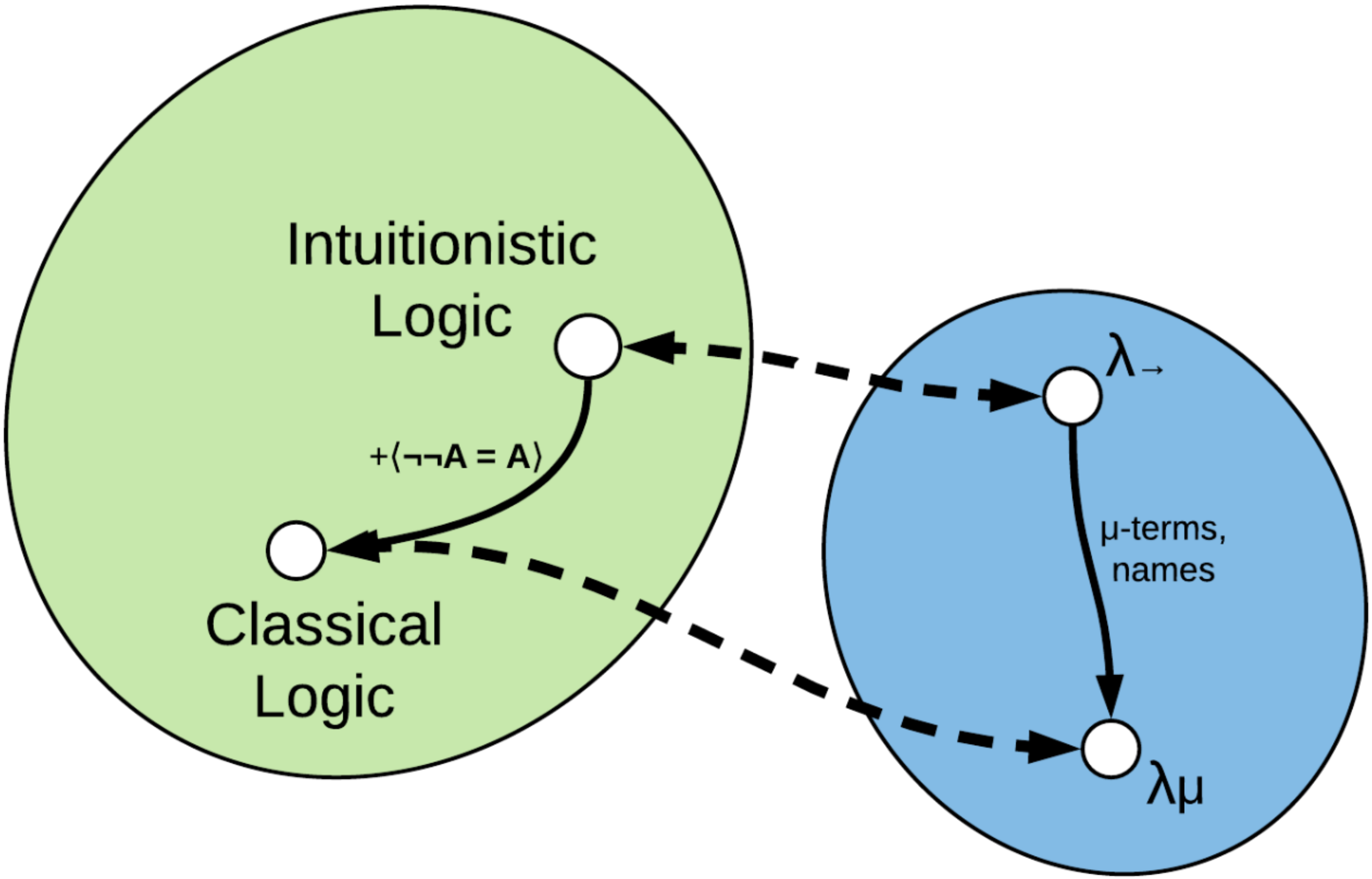
Practically implements **continuations**

Other control mechanisms are easy to construct from it:

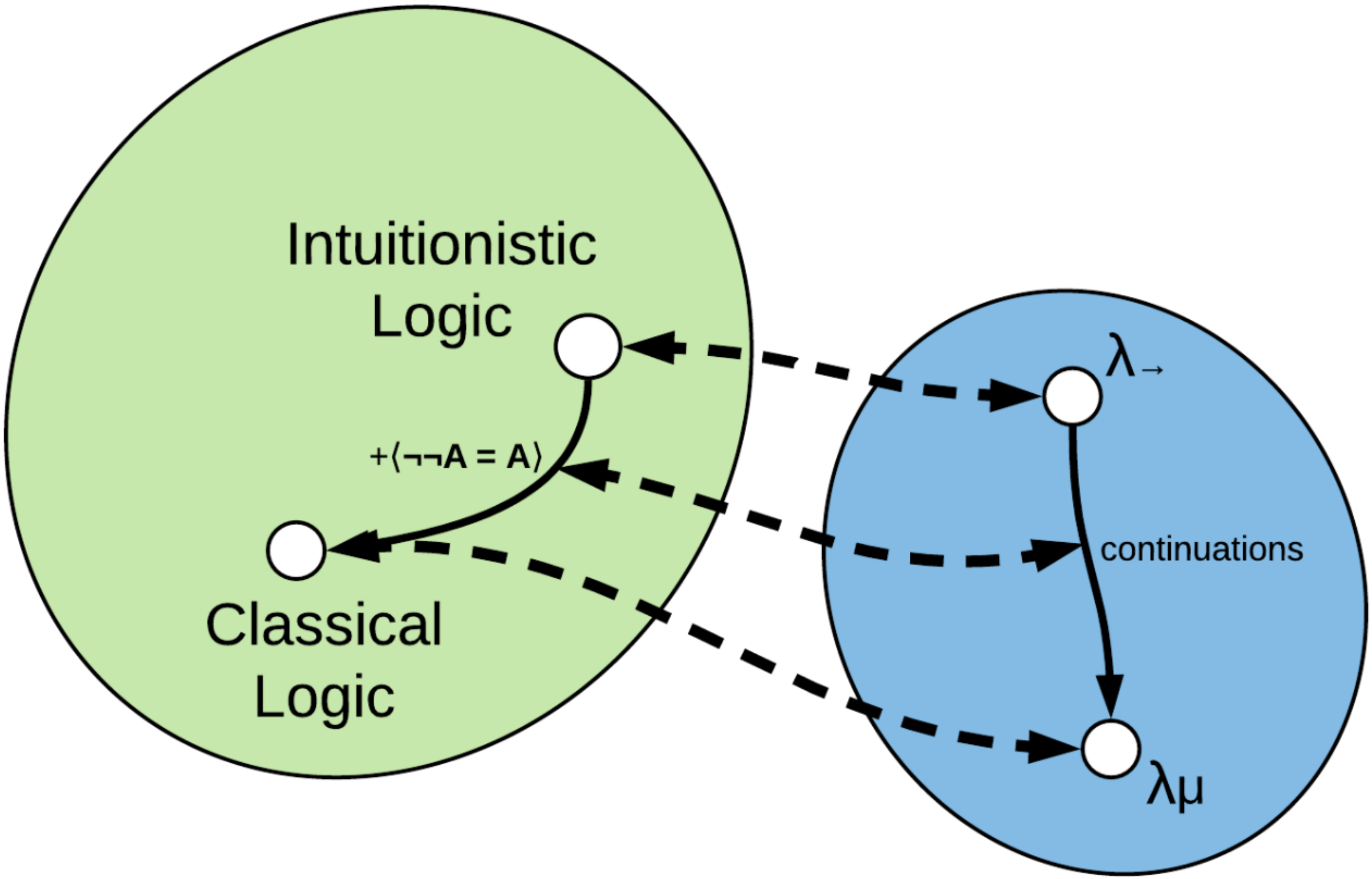
call/cc, throw/catch, reset/shift, ...

Can formalize a proof for **double negation elimination**

Extending the Correspondence



Extending the Correspondence



Outline

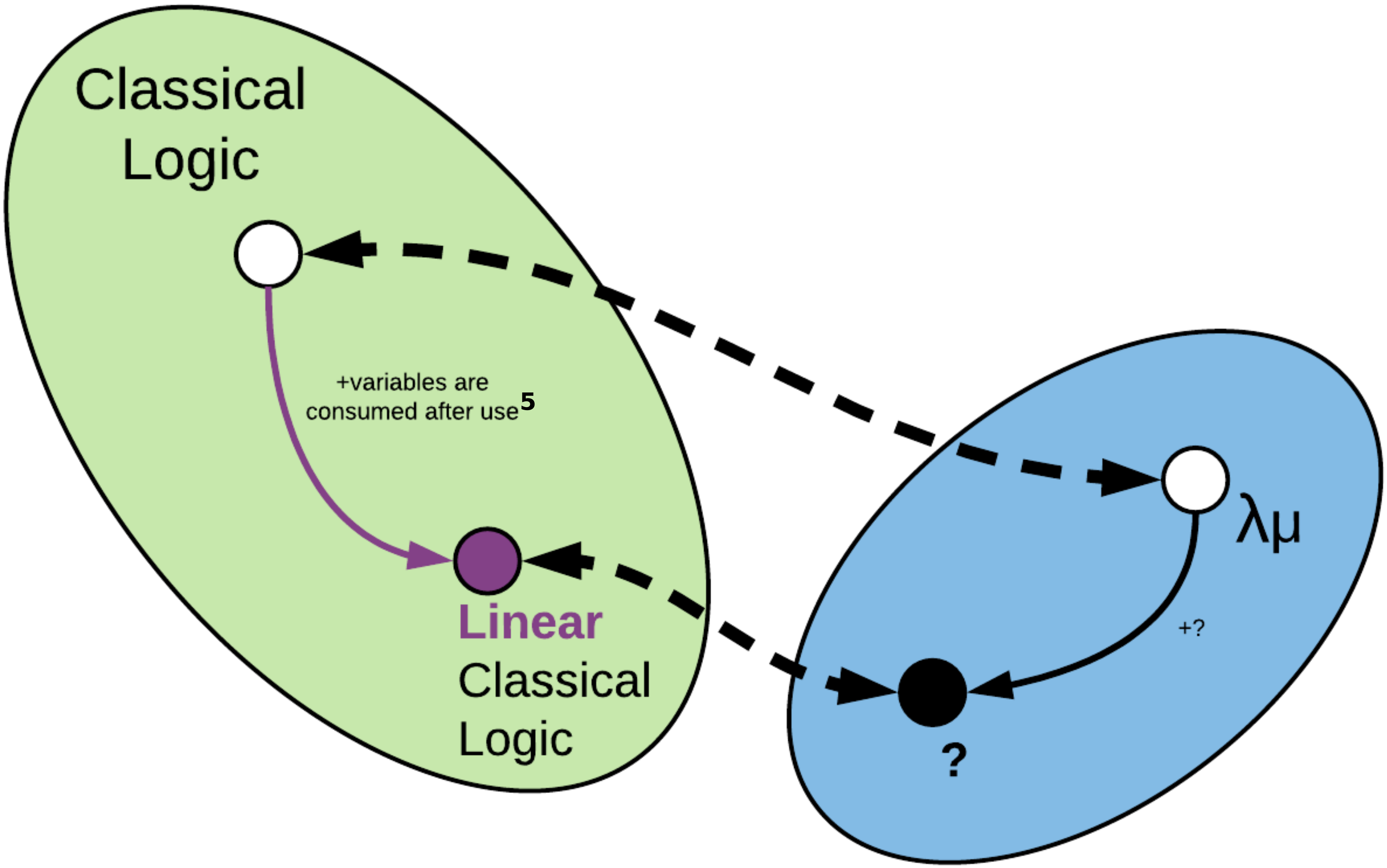
Motivation

Several interesting findings

A more refined perspective

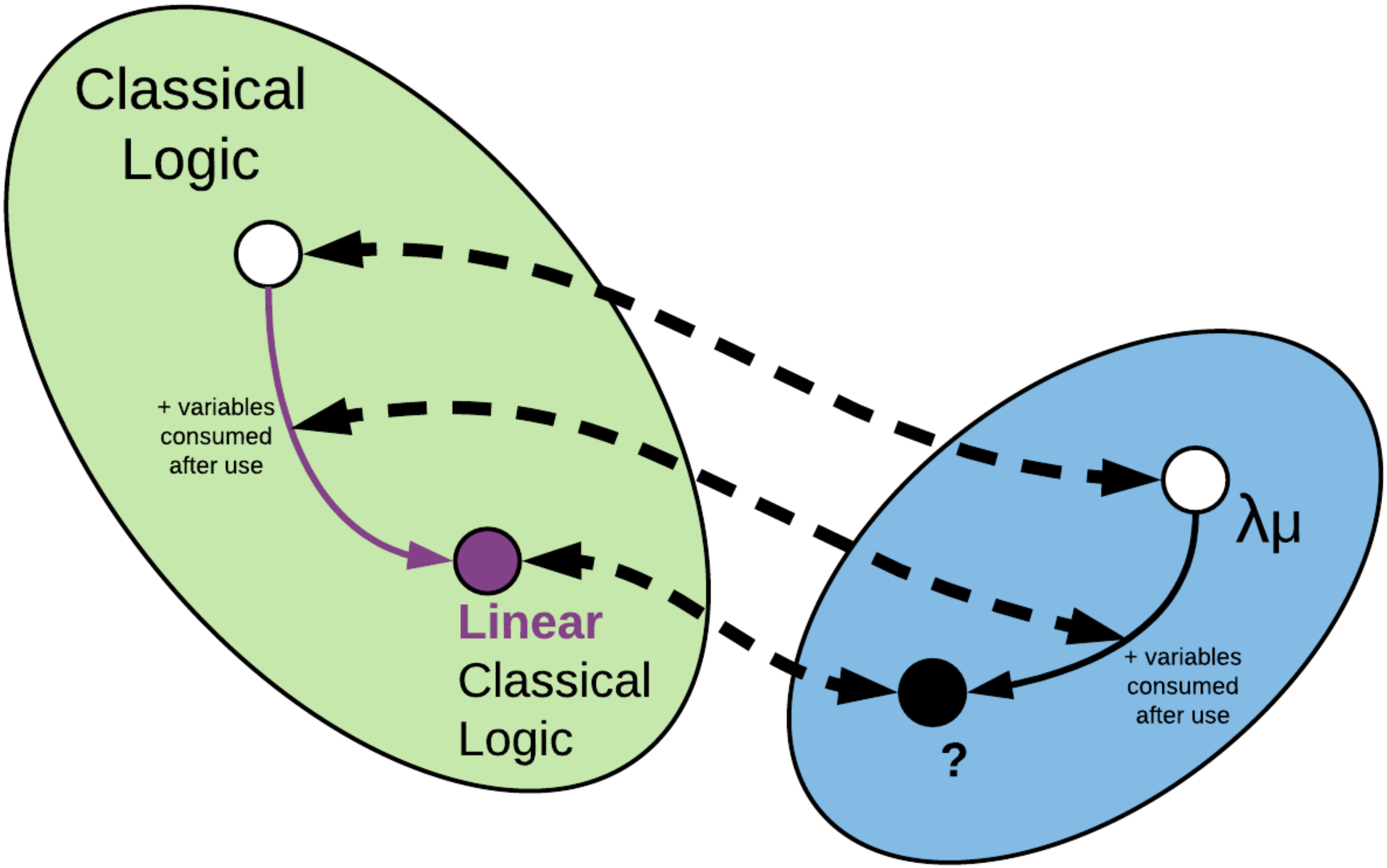
Conclusion

A more Linear Approach



[5] Jean-Yves Girard. 1987. Linear logic. Theor. Comput. Sci. 50, 1 (January 1987), 1-102.

A more Linear Approach



A more Linear Approach

What do we know about this **linear** logic?

Every variable is consumed after usage (can be used only once).

The classical version has the **law of excluded middle**:
Every proposition has its **dual**.

$$P \qquad\qquad\qquad P^{\perp}$$

A more Linear Approach

What do we know about this **linear** logic?

We have *different* operations from classical logic.

$$A : \text{to spend \$1}$$

$$B : \text{to buy a cup of coffee at McDonalds}$$

$$C : \text{to buy a } \textbf{.com} \text{ domain at GoDaddy.com}$$

actually \$0.99 but who's counting

A more Linear Approach

What do we know about this **linear** logic?

We have *different* operations from classical logic.

$$
\begin{aligned}
A \multimap B &: \text{replace \$1 with a cup of coffee} \\
B \otimes C &: \text{get a cup of coffee } \textit{and} \text{ a domain} \\
B \mathbin{\&} C &: \text{choose between coffee } \textit{and} \text{ a domain} \\
B \oplus C &: \text{get either a cup of coffee } \textit{or} \text{ a domain} \\
B \parr C &: \text{get a cup of coffee } \textit{or} \text{ a domain } \textit{in parallel} \\
!A &: \text{of course you can spend more} \\
?A &: \text{why not spend just enough}
\end{aligned}
$$

A more Linear Approach

What do we know about this **linear** logic?

We can model **communication** with these operations[6].

$$
\begin{aligned}
A \multimap B &: \text{replace process A with process B} \\
B \otimes C &: \text{do process B, followed by C} \\
B \mathbin{\&} C &: \text{choose between process B and process C} \\
B \oplus C &: \text{non-deterministically do process B or process C} \\
B \parr C &: \text{do processes B and C } \textit{in parallel} \\
!A &: \text{server response} \\
?A &: \text{client request}
\end{aligned}
$$

[6] Philip Wadler. 2012. Propositions as sessions. In Proceedings of the 17th ACM SIGPLAN international conference on Functional programming(ICFP '12). ACM, New York, NY, USA, 273-286.

A more Linear Approach

Wadler's CP calculus

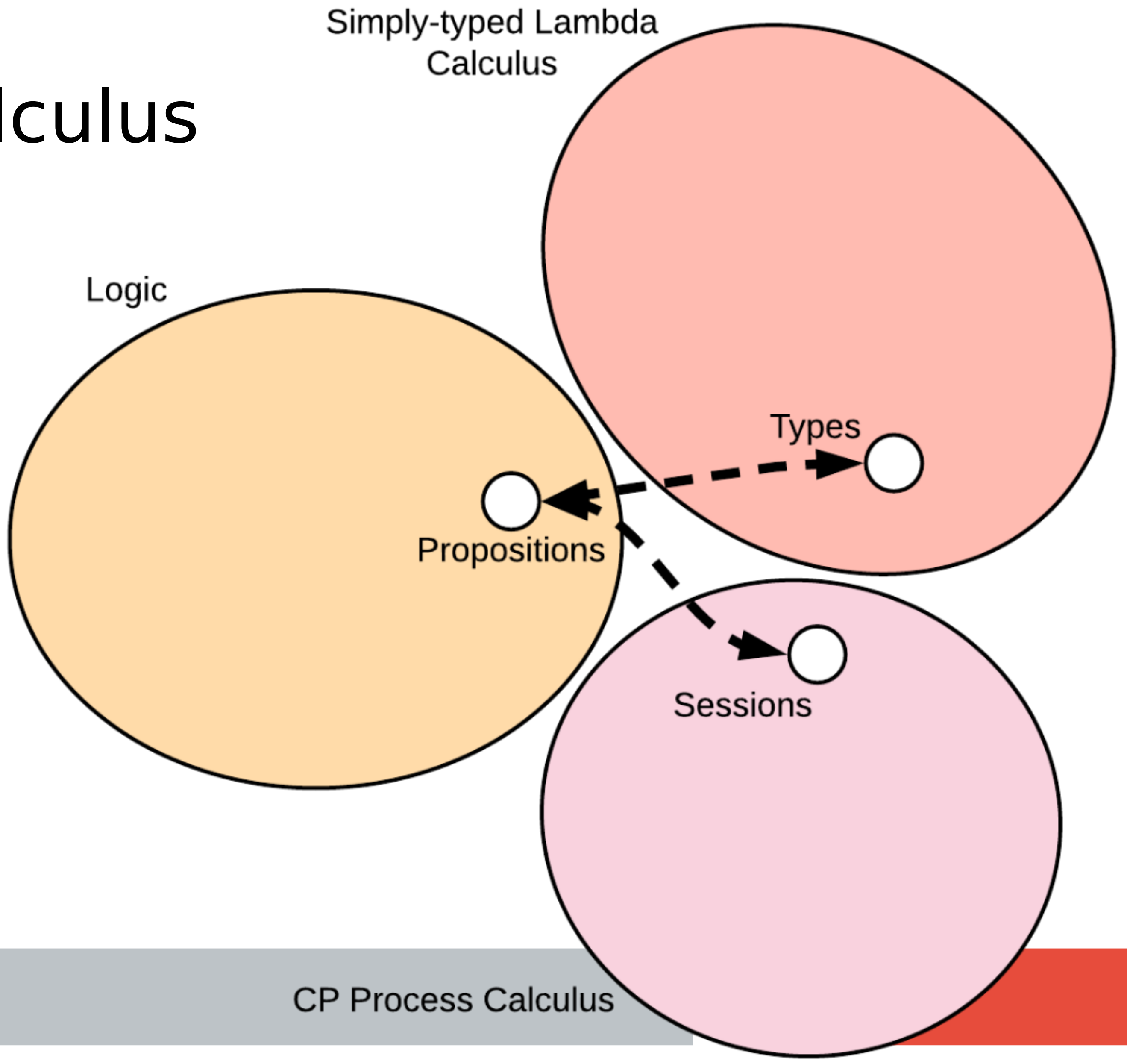
Process calculus: models communication and interactions

Session types: types that, together with their **duals**, define **protocols**

Deadlock safety?

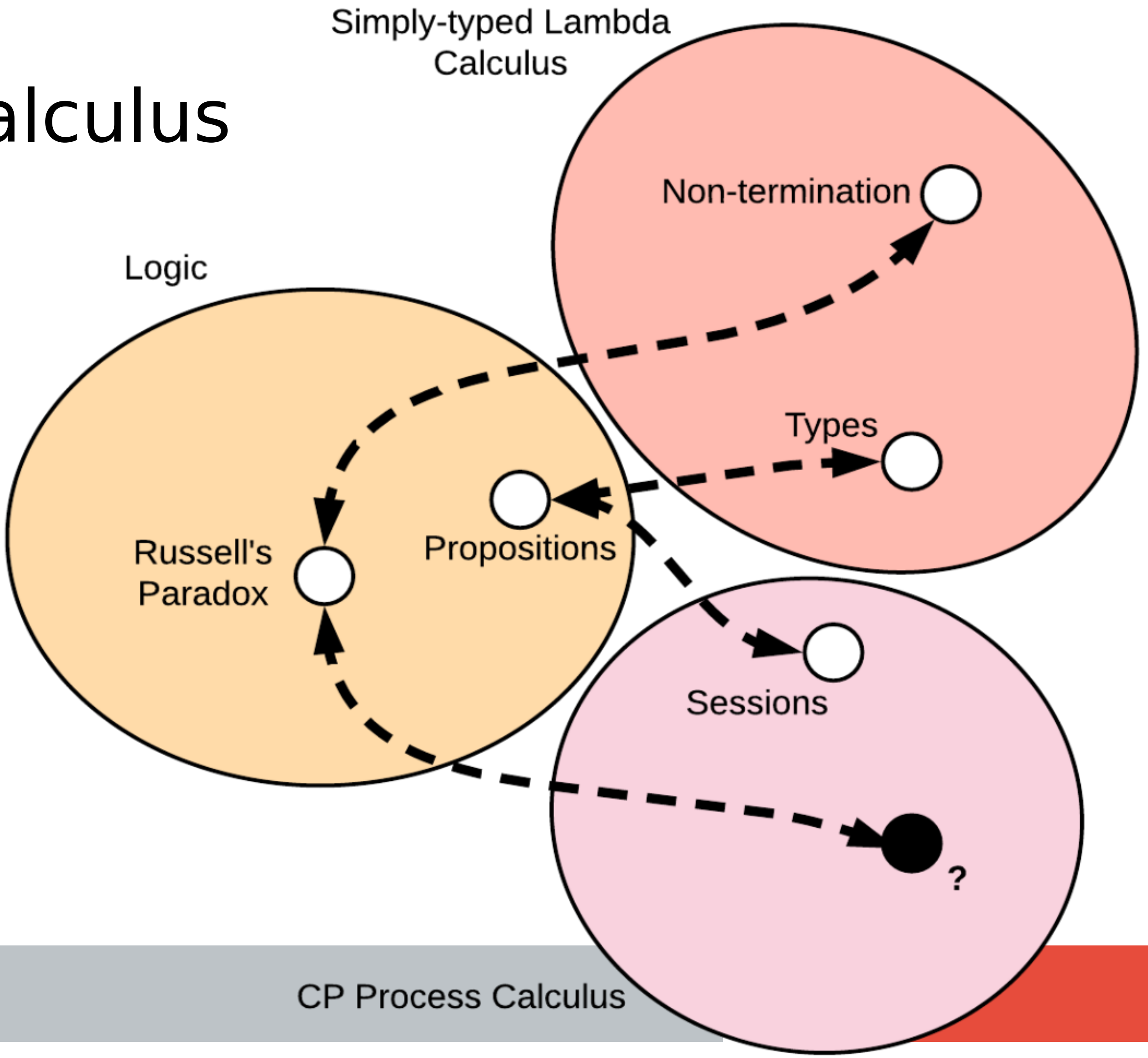
A more Linear Approach

Wadler's CP calculus



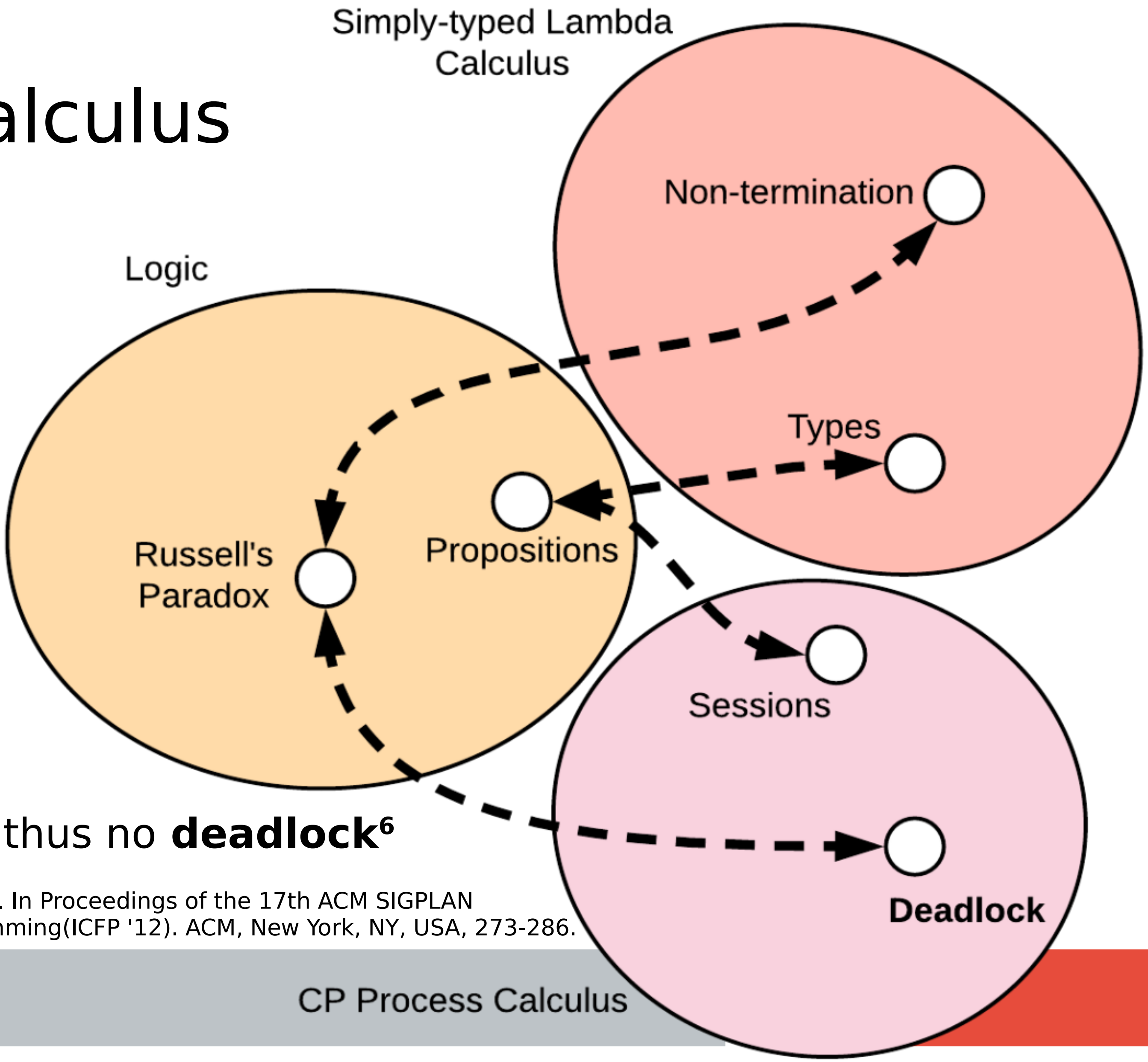
A more Linear Approach

Wadler's CP calculus



A more Linear Approach

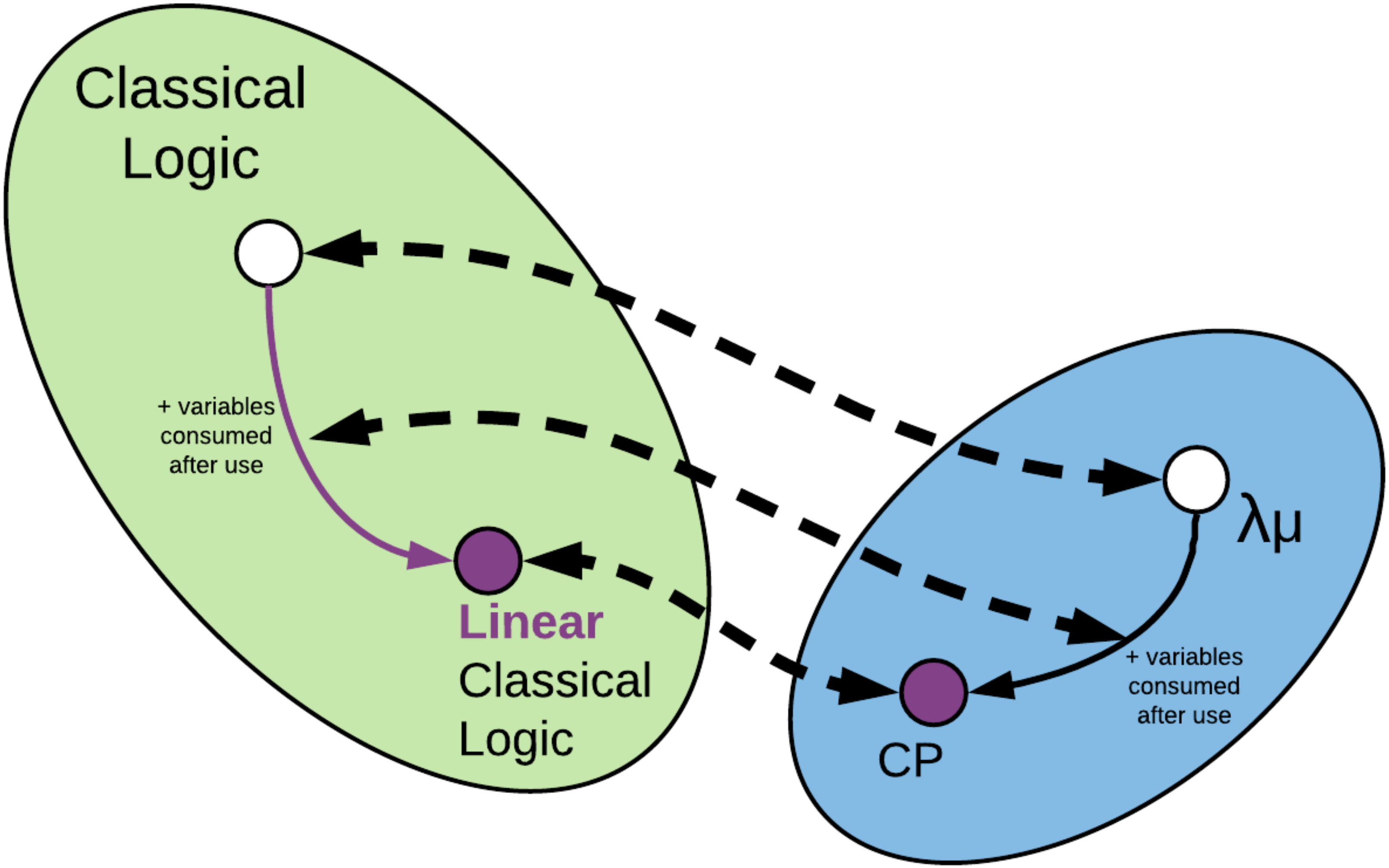
Wadler’s CP calculus



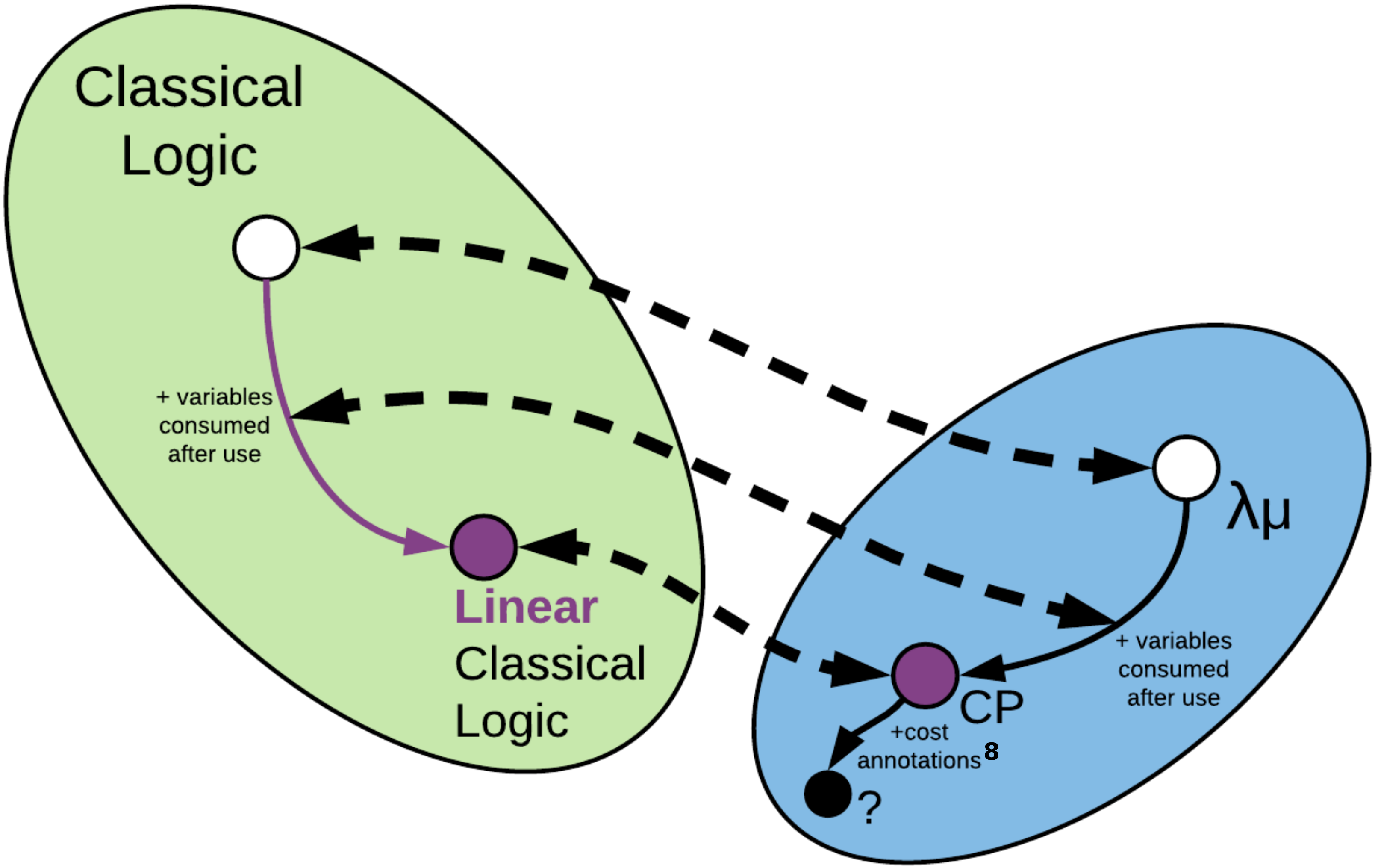
CP is strongly normalizing, thus no **deadlock**[6]

[6] Philip Wadler. 2012. Propositions as sessions. In Proceedings of the 17th ACM SIGPLAN international conference on Functional programming(ICFP '12). ACM, New York, NY, USA, 273-286.

A more Linear Approach

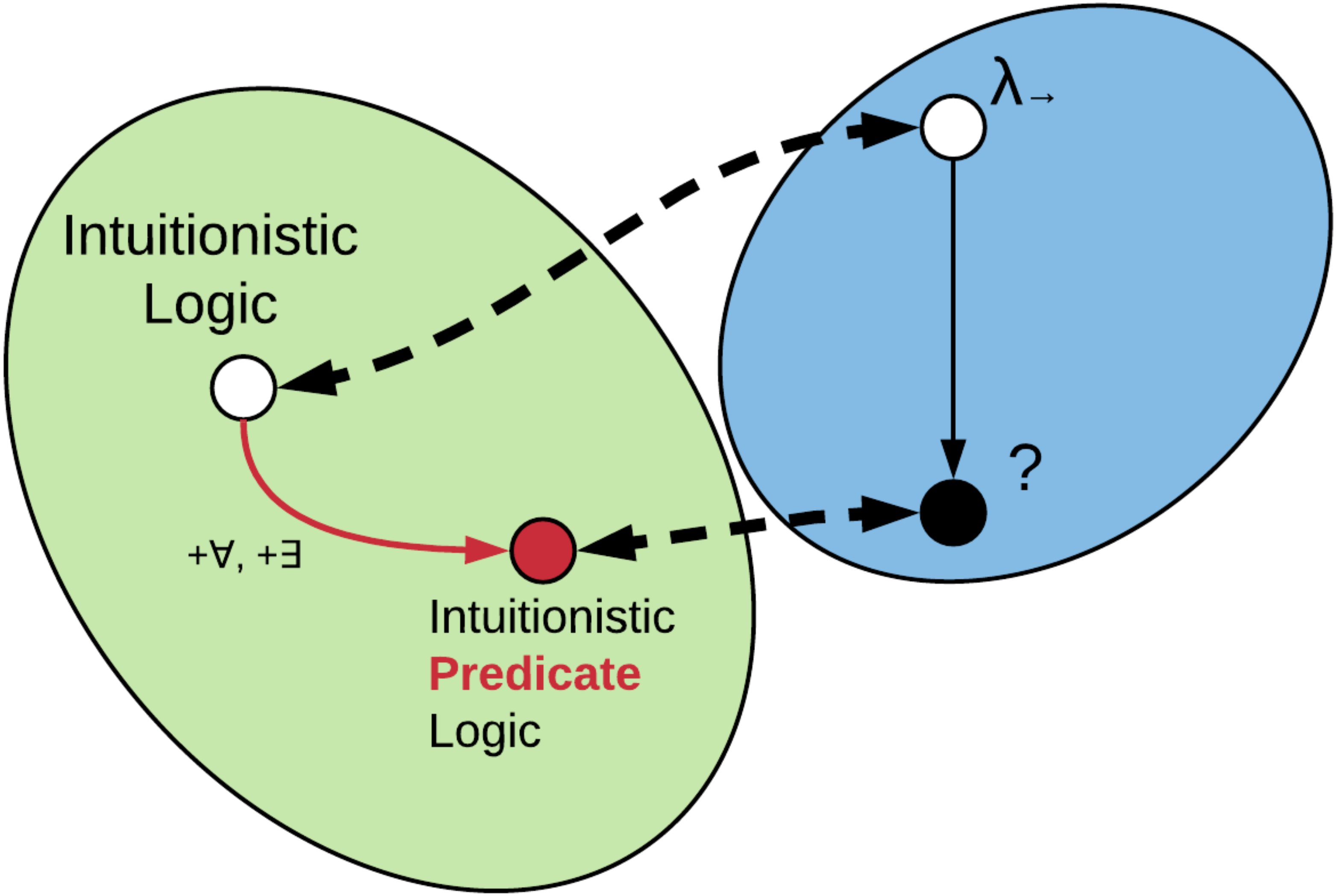


Type-side Extensions?

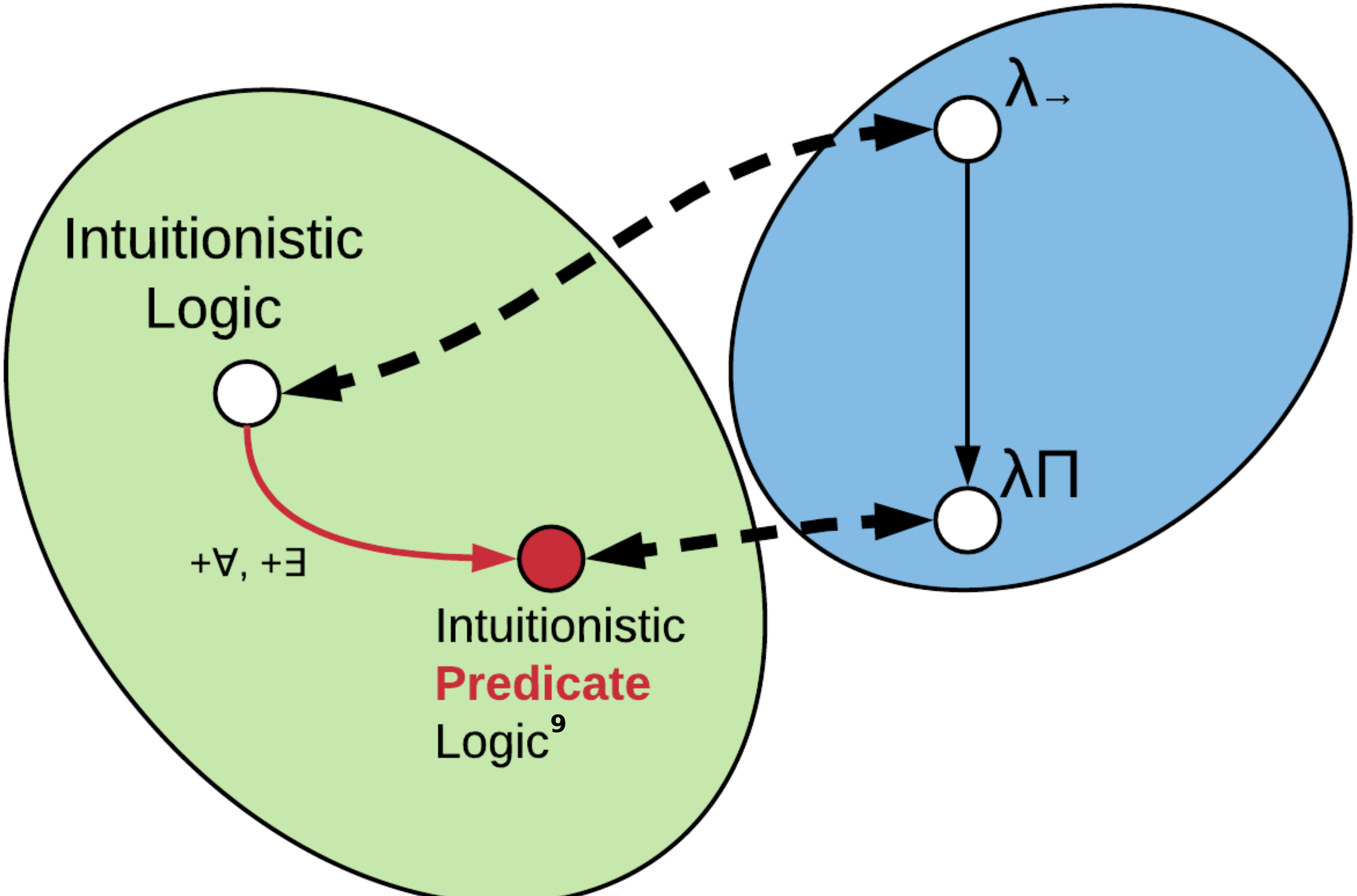


[8] Das, A., Hoffmann, J., & Pfenning, F. (2017). Work Analysis with Resource-Aware Session Types. CoRR, abs/1712.08310.

A Segue into Dependencies



A Segue into Dependencies



[9] Martin-Löf, Per, 1971a, An intuitionistic theory of types, unpublished preprint.

A Segue into Dependencies

λΠ-calculus[10]

Quantification = Types Dependent on Values of other Types

$$\forall x \in A.B(x) \;\equiv \Pi_{x:A}\, B(x)$$
$$\exists x \in A.B(x) \;\equiv \Sigma_{x:A}\, B(x)$$

B can now depend on the value **x**

[10] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming (PPDP '11). ACM, New York, NY, USA, 161-172.

A Segue into Dependencies

λΠ-calculus

Quantification = Types Dependent on Values of other Types

$$\text{append} :: \mathbf{Vector} \to \mathbb{N} \to \mathbf{Vector}$$

A Segue into Dependencies

λΠ-calculus

Quantification = Types Dependent on Values of other Types

$$\mathrm{append} :: \Pi_{n\,:\,\mathbb{N}} \mathbf{Vector}(n) \to \mathbb{N} \to \mathbf{Vector}(n+1)$$

A Segue into Dependencies

λΠ-calculus

Quantification = Types Dependent on Values of other Types

$$\mathrm{append} :: \boxed{\Pi_{n\,:\,\mathbb{N}}} \mathbf{Vector}(n) \to \mathbb{N} \to \mathbf{Vector}(n+1)$$

A Segue into Dependencies

λΠ-calculus

Quantification = Types Dependent on Values of other Types

$$\text{append} :: \Pi_{n\,:\,\mathbb{N}} \boxed{\mathbf{Vector}(n)} \to \mathbb{N} \to \mathbf{Vector}(n+1)$$

A Segue into Dependencies

λΠ-calculus

Quantification = Types Dependent on Values of other Types

$$\text{append} :: \Pi_{n\,:\,\mathbb{N}} \mathbf{Vector}(n) \to \mathbb{N} \to \boxed{\mathbf{Vector}(n+1)}$$

A Segue into Dependencies

λΠ-calculus

Quantification = Types Dependent on Values of other Types

$$\text{append} :: \Pi_{n\,:\,\mathbb{N}} \mathbf{Vector}(n) \to \mathbb{N} \to \mathbf{Vector}(n+1)$$

The dependency relation doesn't have to be this simple...

Information-flow Types

Data dependent on **security lattices**[11]

$$\mathbf{Type}^{\mathrm{security}}$$

$$\mathbf{Int}^{\top}$$

$$\mathbf{Profile}^{\mathrm{level(user)}}$$

$$\mathbf{String}^{\perp}$$

[11] Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15). ACM, New York, NY, USA, 317-328.

Information-flow Types

Data dependent on **security lattices**

$$\text{get\_profile} :: \mathbf{Id}^{\perp} \rightarrow \mathbf{Profile}^{\text{level}\boxed{(\text{user})}}$$

$$\text{get\_profile} :: \mathbf{Id}^{\perp} \rightarrow \boxed{\Sigma_{\text{user}:\mathbf{User}}} . \mathbf{Profile}(\text{user})^{\text{level}(\text{user})}$$

Information-flow Types

Data dependent on **security lattices**

Security labels are passed through typing judgements, but such type systems can be *too* strict

Access-policy synthesis[12]: weaker system, and the results of failed verifications can be used to identify where sensitive information leaked and automatically repair it

[12] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. ACM Trans. Program. Lang. Syst. 35, 2, Article 6 (July 2013), 41 pages.

Information-flow Types

Proof assistants can be used to check permissions while typing out a dependently-typed program[13]!

```
solve : ∀{T} (s : Maybe T) → { p : Check isSome(s) } → T

...

proof? : Maybe (Proof Γ MayRead(user, "secret.txt"))
proof? = solve (proveToDepth 15)
```

[13] Jamie Morgenstern and Daniel R. Licata. 2010. Security-typed programming within dependently typed programming. In Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10). ACM, New York, NY, USA, 169-180.

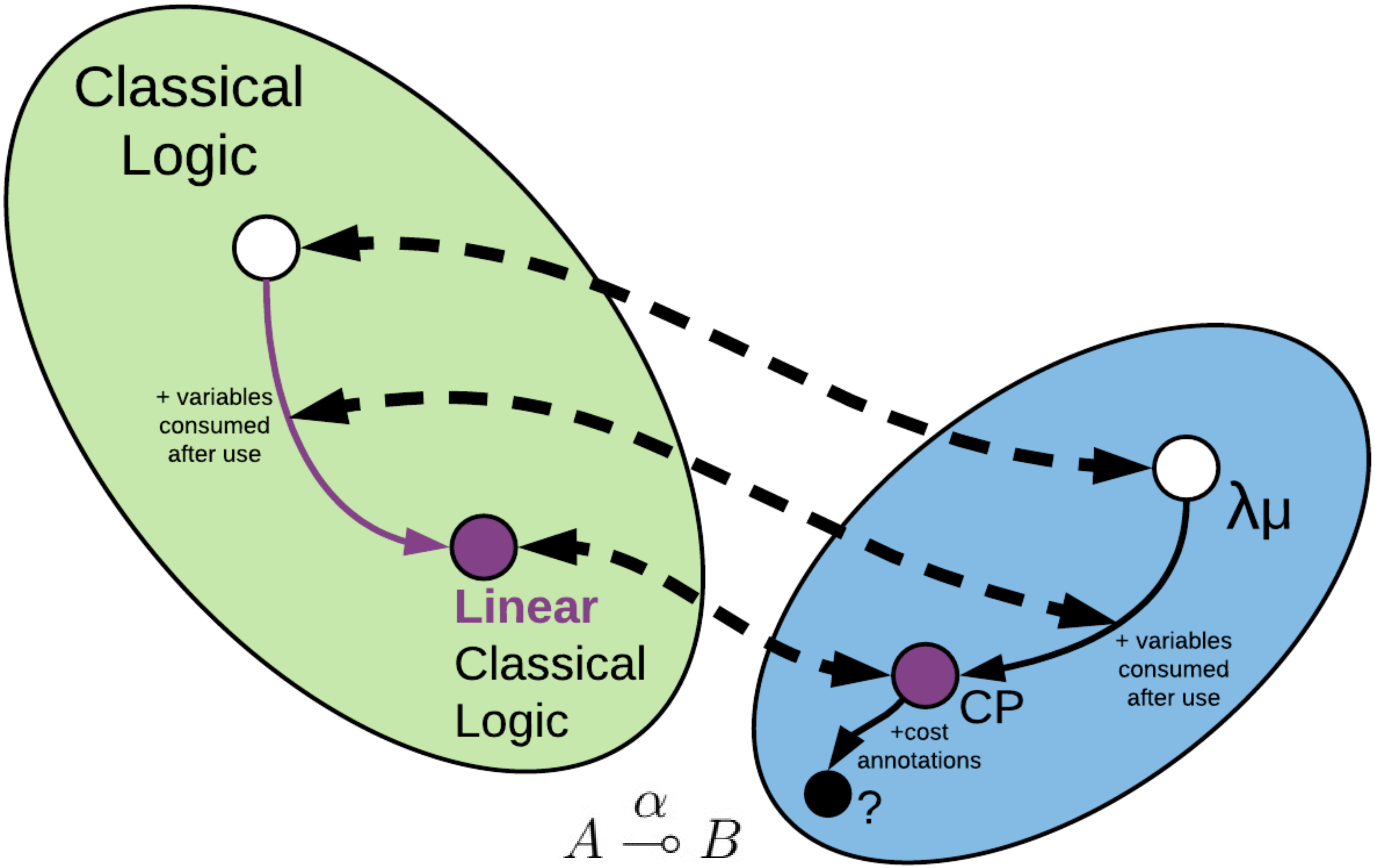
Information-flow Types

Proof assistants can be used to check permissions while typing out a dependently-typed program!

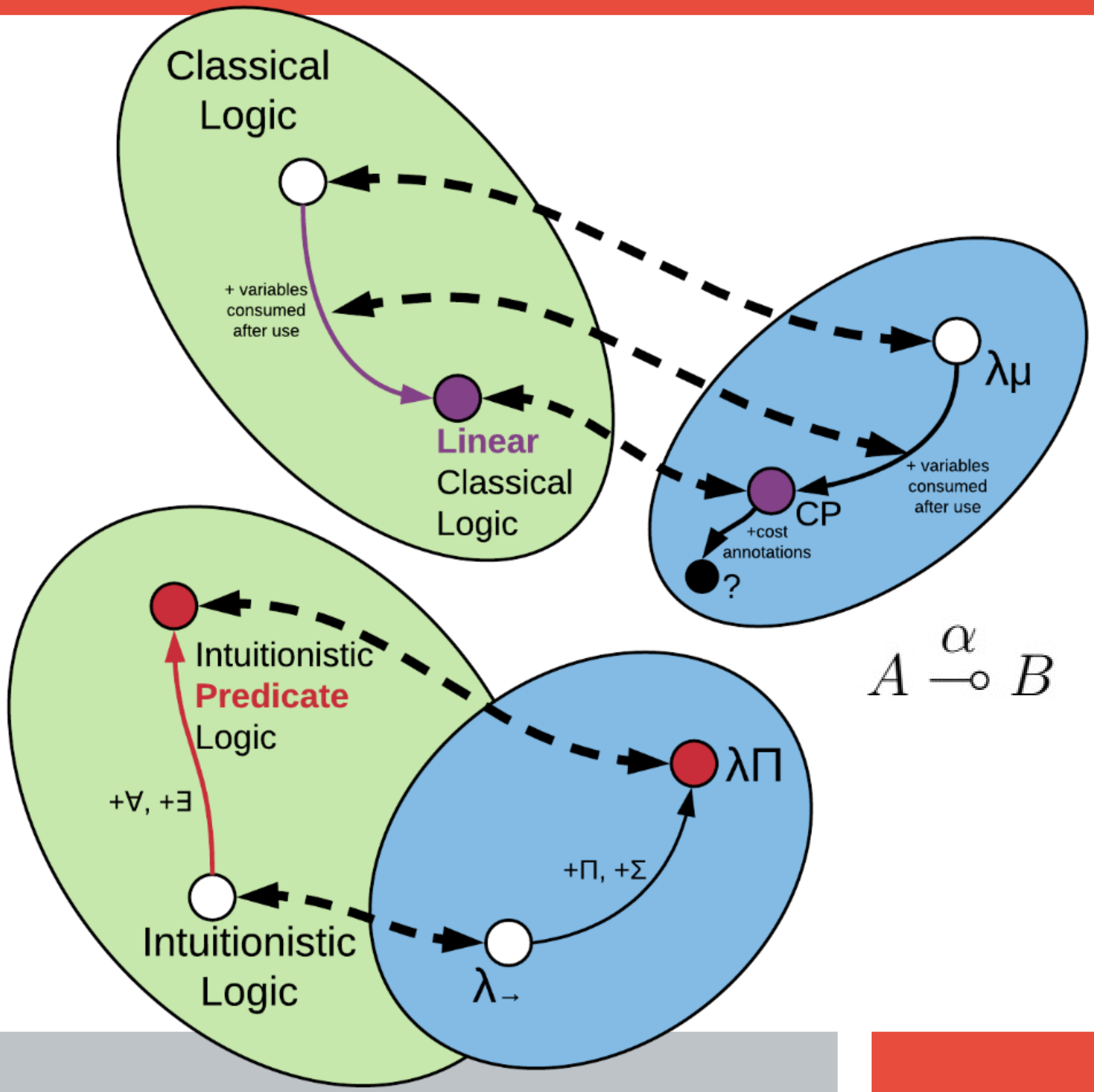
This is an interactive static method of information-flow handling.

```
solve : ∀{T} (s : Maybe T) → { p : Check isSome(s) } → T

...

proof? : Maybe (Proof Γ MayRead(user, "secret.txt"))
proof? = solve (proveToDepth 15)
```

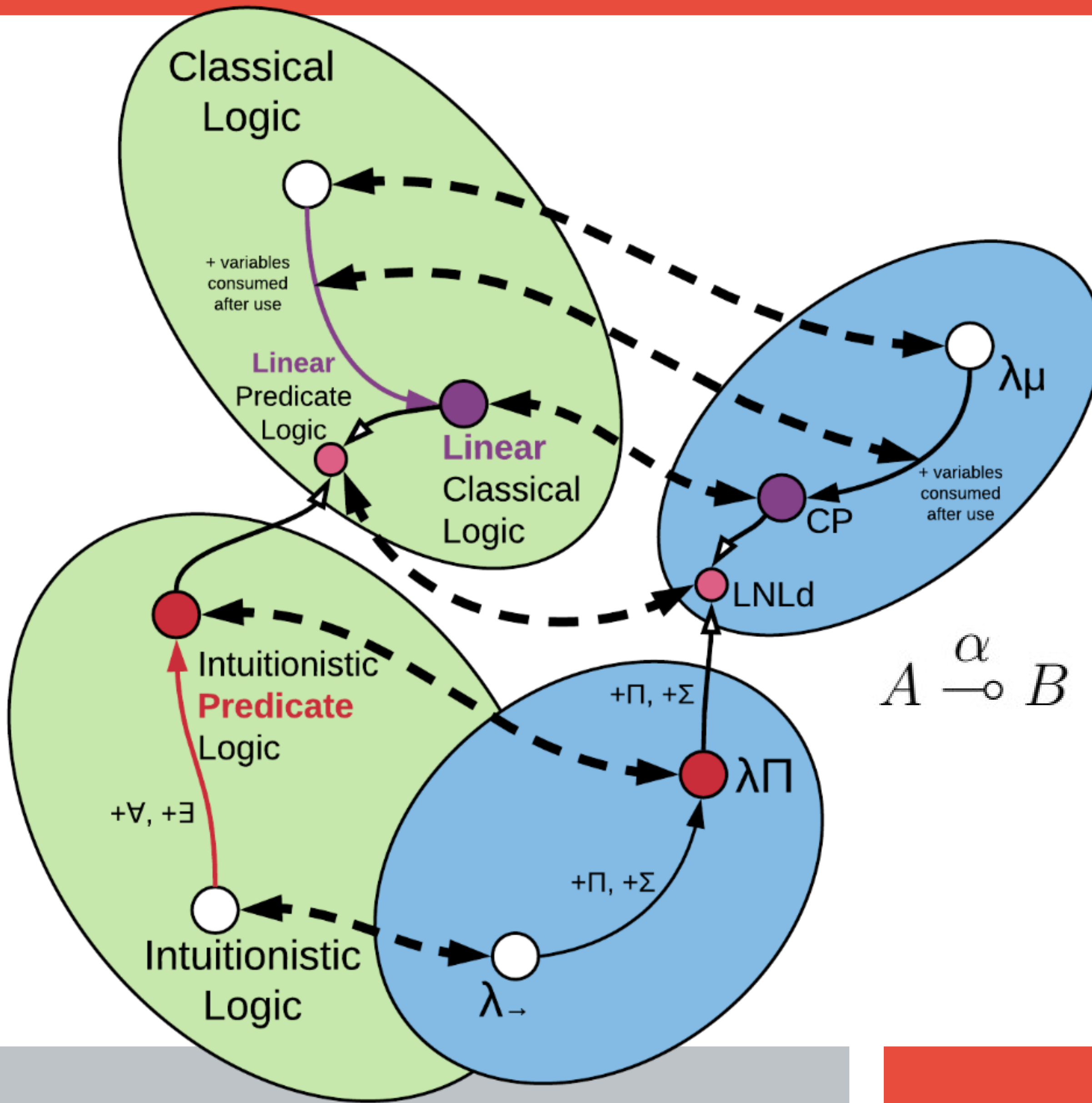
Are we there yet?



Are we there yet?



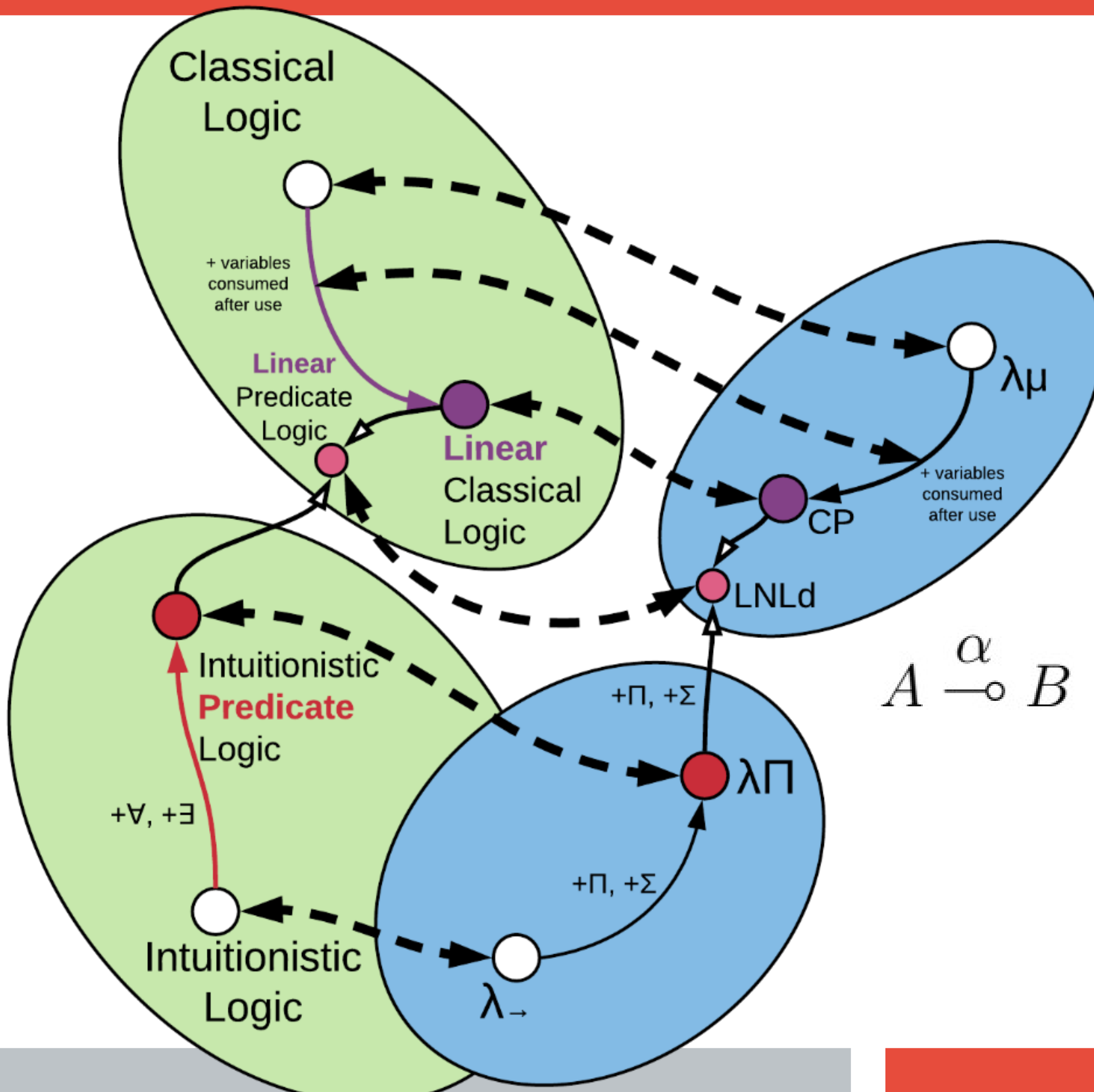
Are we there yet?



[14] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15). ACM, New York, NY, USA, 17-30.

Are we there yet?

We did it, but...
Overkill much?



$$A \overset{\alpha}{\multimap} B$$

Outline

Motivation

Several interesting findings

A more refined perspective

Conclusion

Extending Type Systems

Augmenting type systems:

changing them so that they're more *useful* is *natural*.

Type systems as a basis for analyses[15]:

- used as a starting point for other analyses
  (method inlining, redundant load elimination, etc.)

- small extensions to type systems yield large benefits
  (encapsulation checking, race detection, etc.)

- changes along known Curry-Howard lines
  (memory management, regions, linear types, etc.)

[15] Jens Palsberg. 2001. Type-based analysis and applications. In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01). ACM, New York, NY, USA, 20-27. DOI=http://dx.doi.org/10.1145/379605.379635

Type Effects[16]

Type-and-Effects: enriched type-system, with annotations for describing intentional aspects of dynamic behaviour[17]

```
def saveToFile(f: File, d: Data): Unit =
{
  write(f, d);  // error: f is closed
  close(f);     // error: f is closed
}
```

[16] Nielson F., Nielson H.R., Hankin C. (1999) Type and Effect Systems. In: Principles of Program Analysis. Springer, Berlin, Heidelberg
[17] Yitzhak Mandelbaum, David Walker, and Robert Harper. 2003. An effective theory of type refinements. In Proceedings of the eighth ACM SIGPLAN international conference on Functional programming (ICFP '03). ACM, New York, NY, USA, 213-225.

Type Effects

Type-and-Effects: enriched type-system, with annotations for describing intentional aspects of dynamic behaviour

```
def saveToFile(f: File, d: Data; closed(f)): (Unit; closed(f)) =
{
                  // f starts out closed
  open(f);        // f is now opened
  write(f, d);    // okay, f stays opened
  close(f);       // f is now closed
                  // f finishes closed
}
```

Type Effects

Type-and-Effects: enriched type-system, with annotations for describing intentional aspects of dynamic behaviour

Assume input effects

```
def saveToFile(f: File, d: Data; closed(f)): (Unit; closed(f)) =
{
                 // f starts out closed
  open(f);       // f is now opened
  write(f, d);   // okay, f stays opened
  close(f);      // f is now closed
                 // f finishes closed
}
```

Type Effects

Type-and-Effects: enriched type-system, with annotations for describing intentional aspects of dynamic behaviour

Assert output effects

```
def saveToFile(f: File, d: Data; closed(f)): (Unit  closed(f)) =
{
                // f starts out closed
  open(f);      // f is now opened
  write(f, d);  // okay, f stays opened
  close(f);     // f is now closed
                // f finishes closed
}
```

Type Effects

Type-and-Effects: enriched type-system, with annotations for describing intentional aspects of dynamic behaviour

Similar to Hoare logic, but applicable to higher-order programs and with decidable type-checking

Type Refinements

Type Refinements: predicates over type (every refined type is a subtype of the original type)

Liquid types[18]:
*L*ogically *Q*ualified *D*ata Types, combine Hindley-Milner type inference with predicate abstraction to infer dependent types

Give type-safety proofs for *overapproximation*, *preservation* and *progress*.

[18] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). ACM, New York, NY, USA, 159-169.

Type Refinements

$$\text{average} :: [\mathbf{Int}] \to \mathbf{Int}$$
$$\text{average } xs = \text{sum } xs \text{ div length } xs$$

$$\text{average} :: \{L@[\mathbf{Int}] \mid \text{length } L > 0\} \to \mathbf{Int}$$
$$\text{average } xs = \text{sum } xs \text{ div length } xs$$

Type Refinements

Type Refinements: predicates over type (every refined type is a subtype of the original type)

Specifications generally limited to decidable fragments of underlying logic, with predefined *refined operations*

$$\text{average} :: \{L@[\mathbf{Int}] \mid \boxed{\text{length}}\ L > 0\} \to \mathbf{Int}$$

$$\text{average}\ xs = \text{sum}\ xs\ \text{div}\ \text{length}\ xs$$

Type Refinement Systems

Type Refinements: actually more general than they seem.

...through a categorical lens

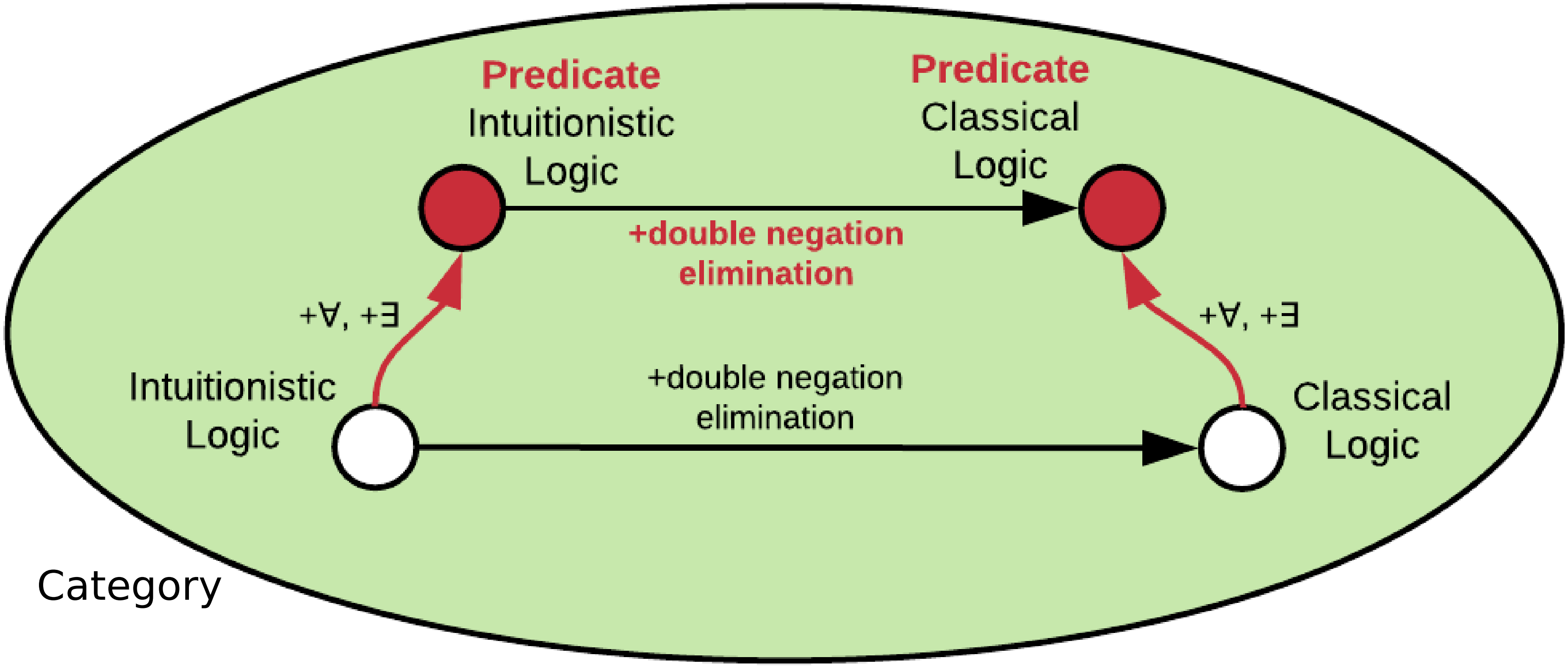
Functors are Type Refinement Systems[20]

Functors: mappings that preserve structure between two categories

[20] Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15). ACM, New York, NY, USA, 3-16.

Type Refinement Systems

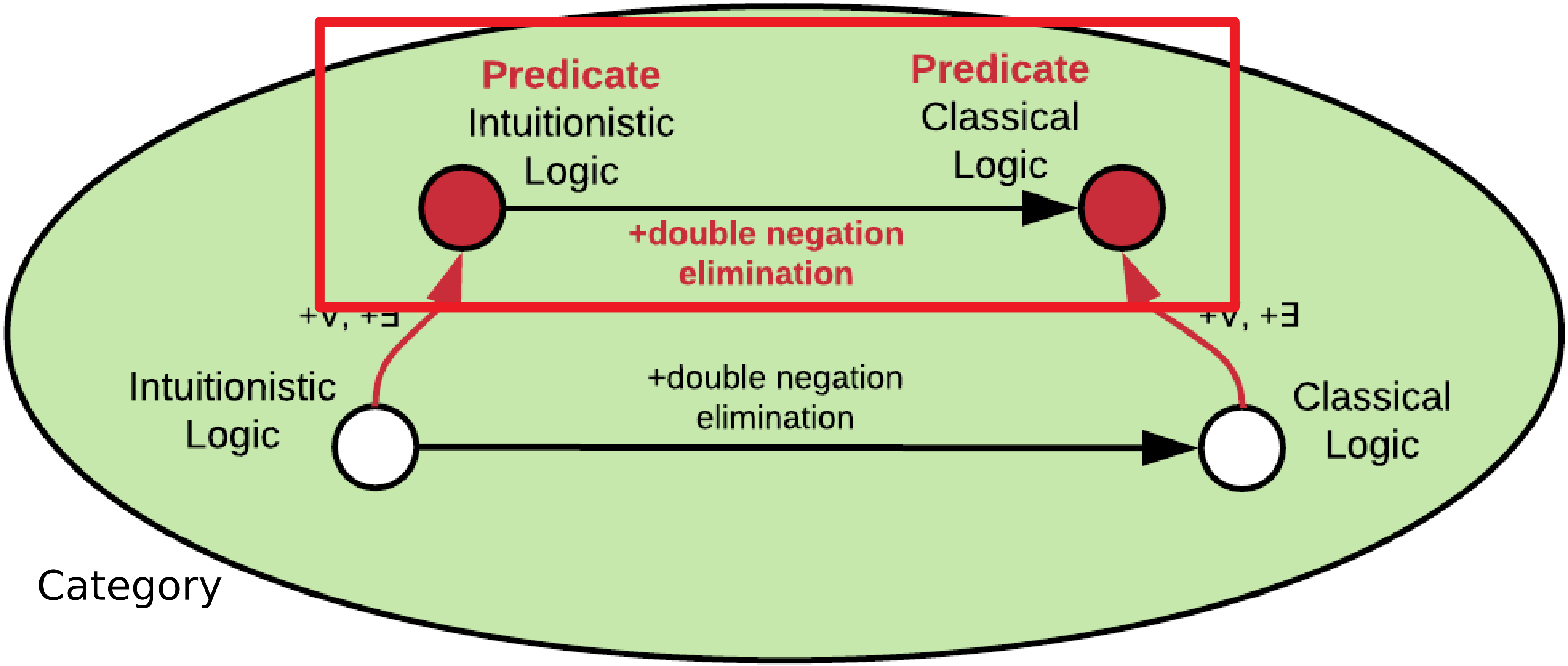
Functor preserves mapping between two categories



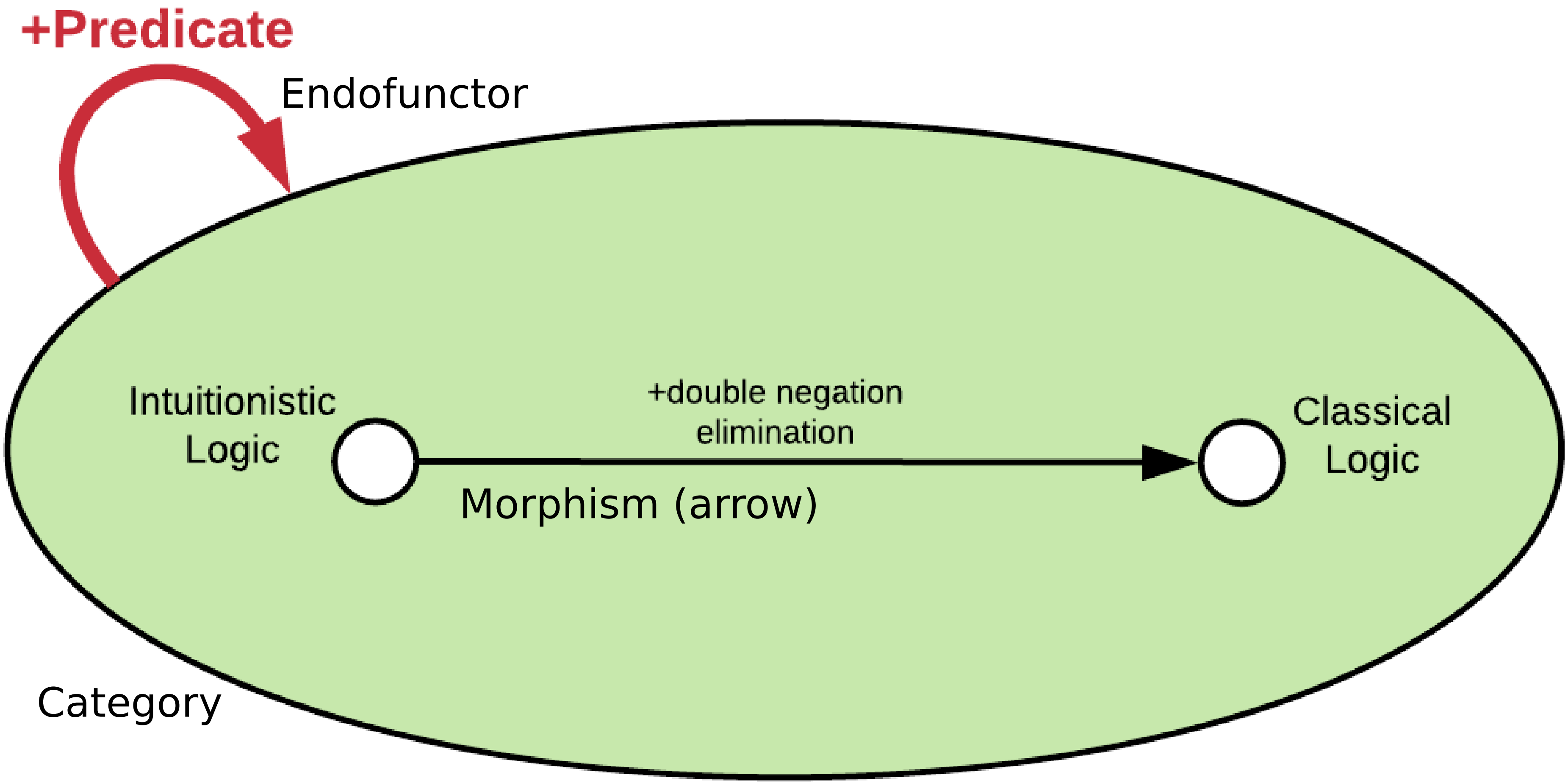
Category

Type Refinement Systems

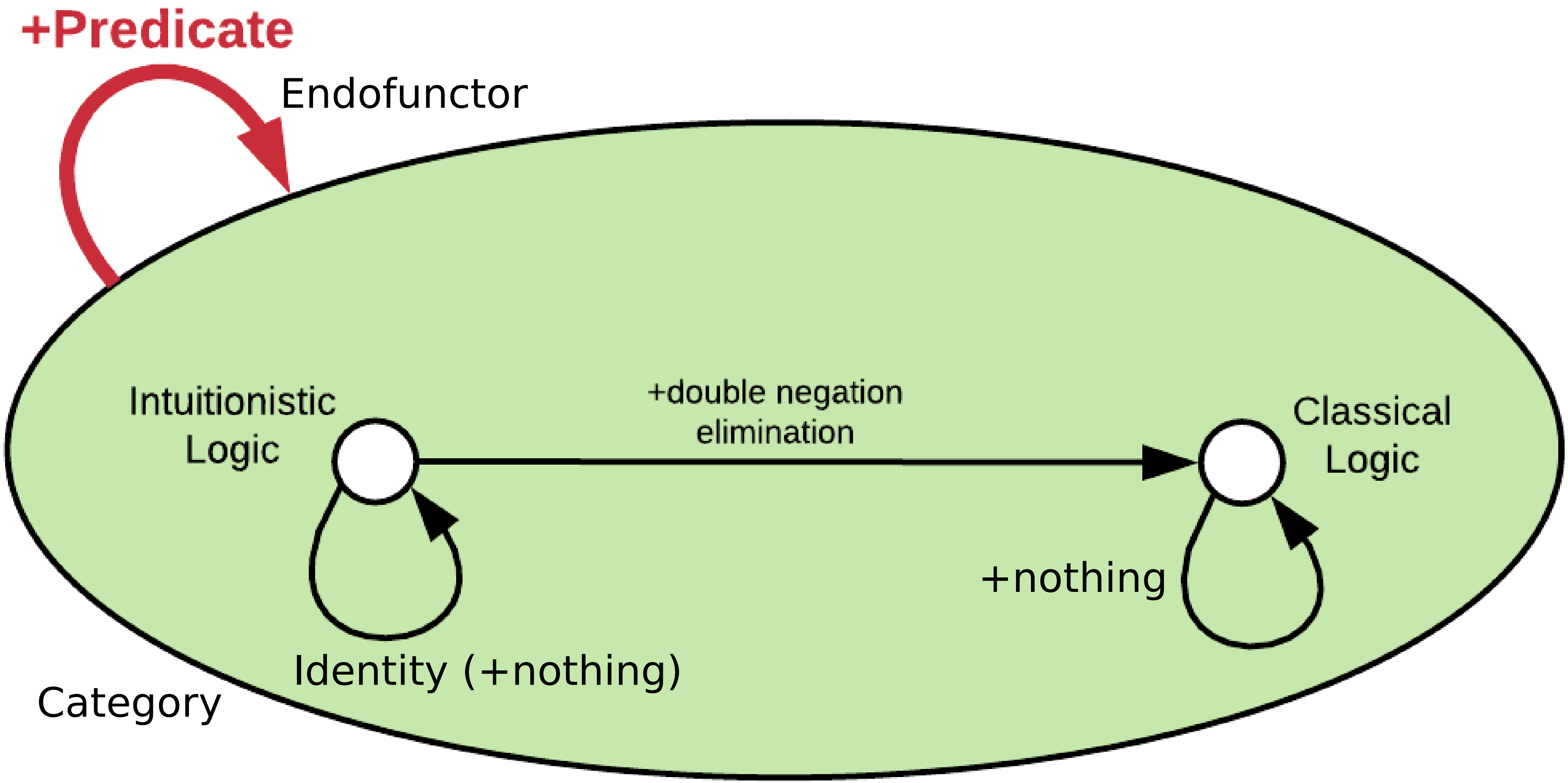
Functor preserves mapping between two categories



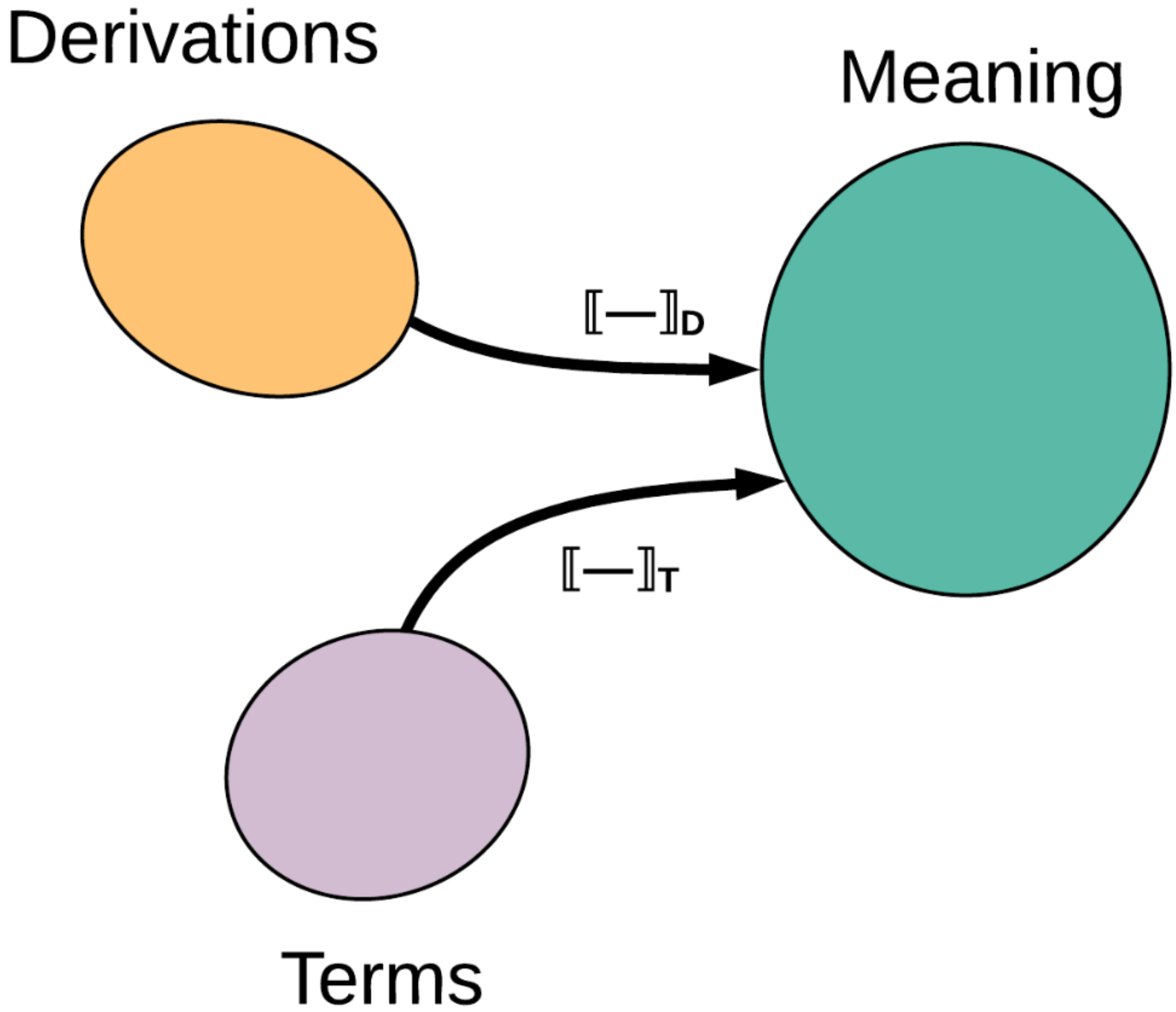
Type Refinement Systems



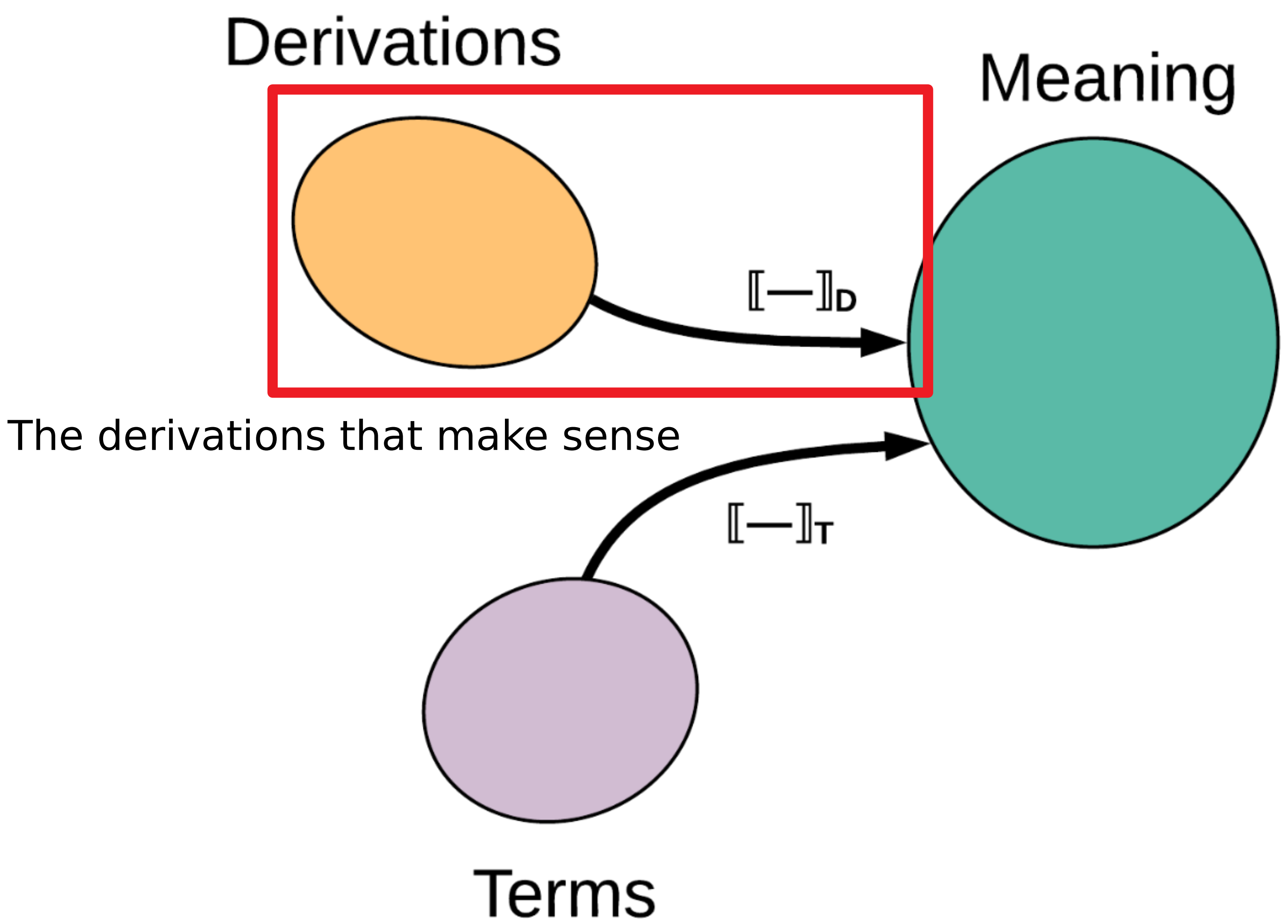
Type Refinement Systems



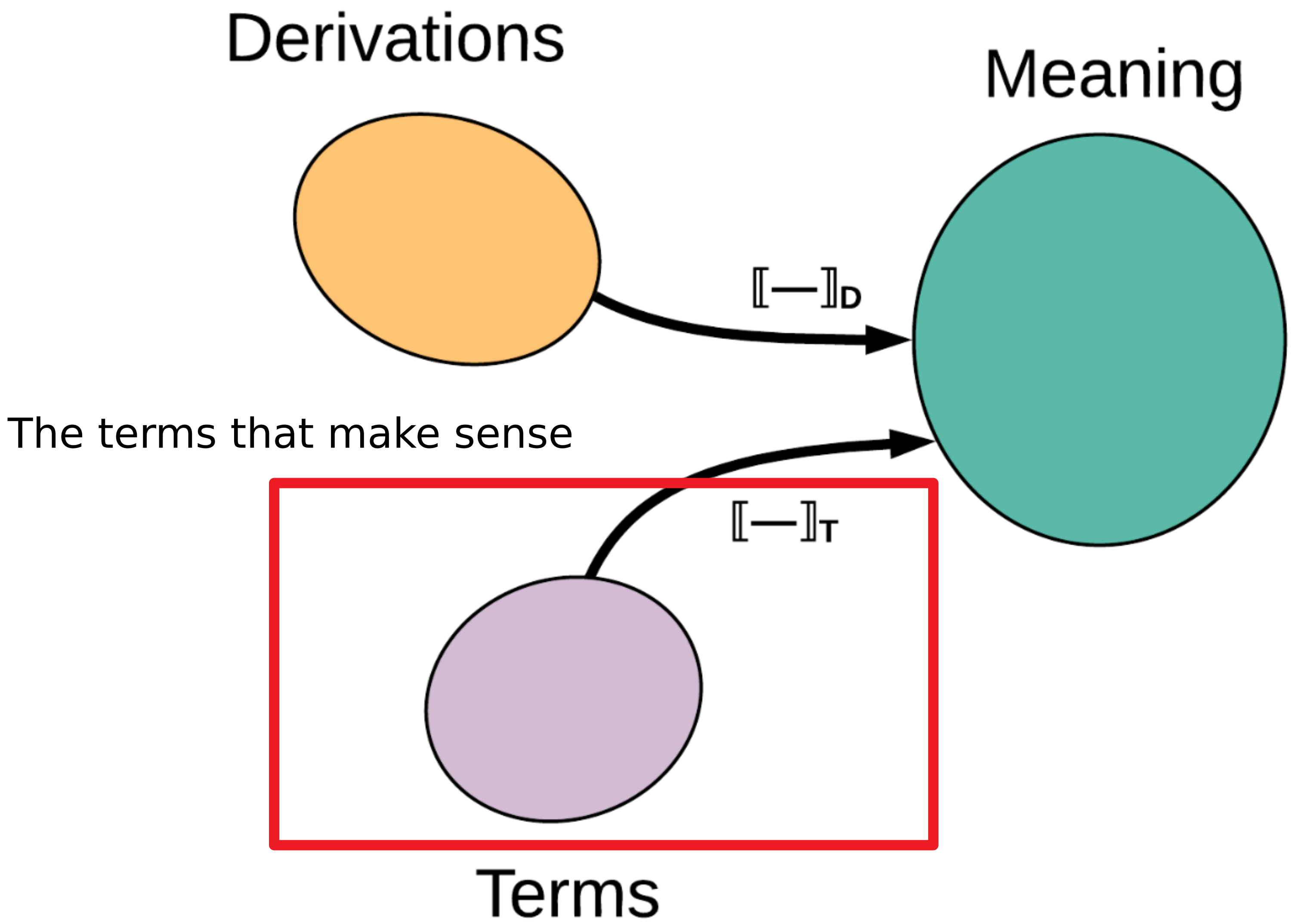
Type Refinement Systems



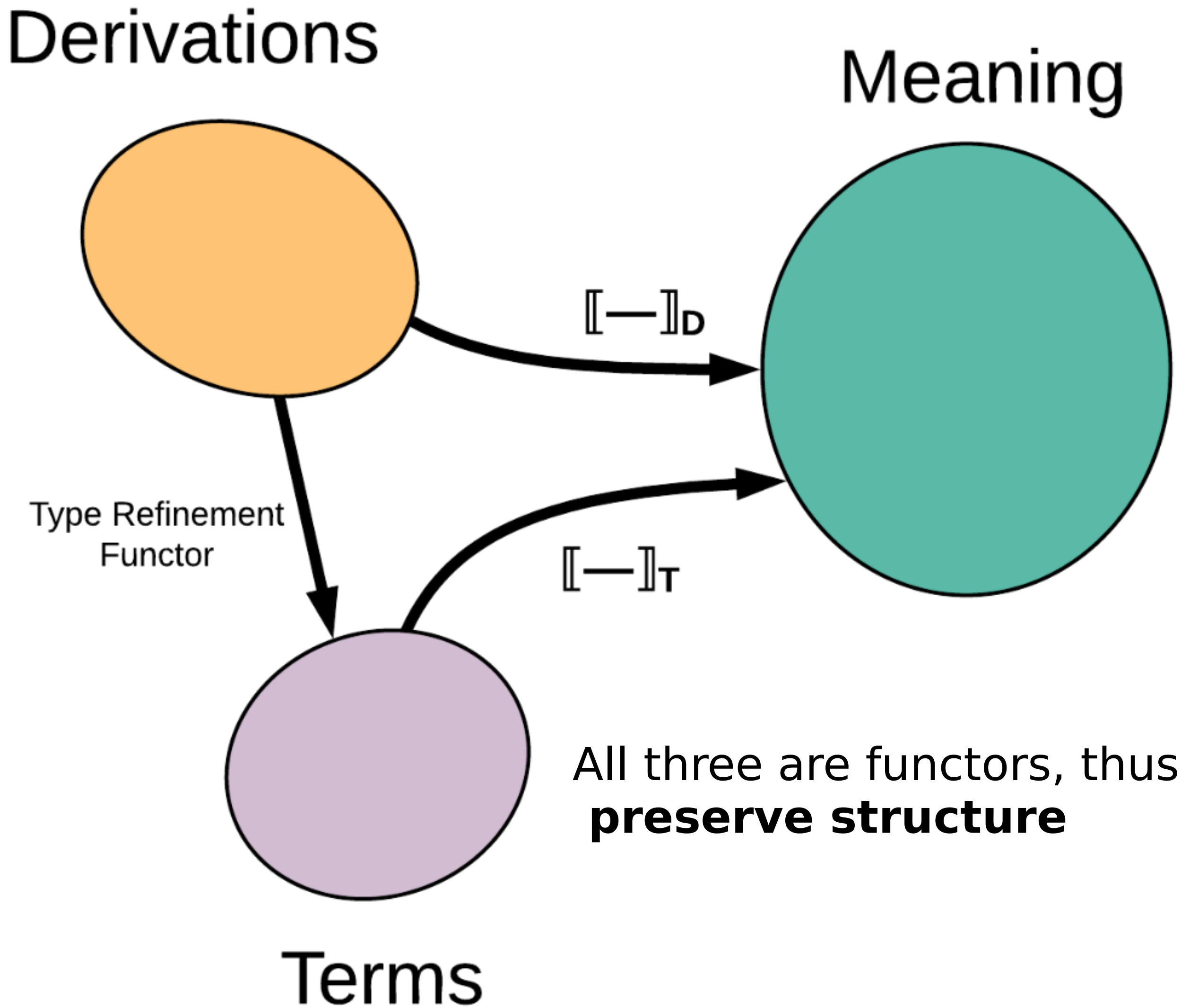
Type Refinement Systems



Type Refinement Systems

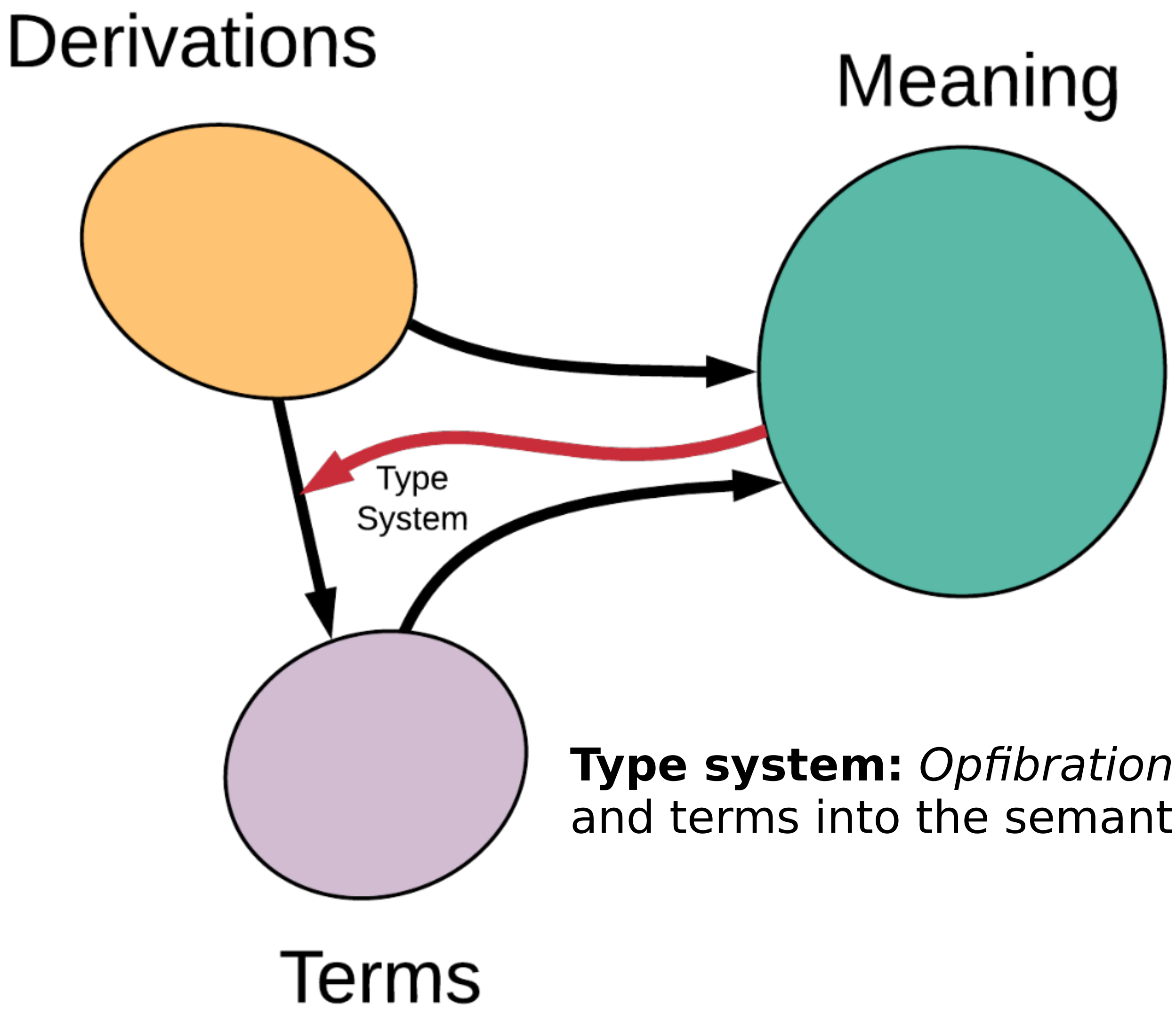


Type Refinement Systems



All three are functors, thus they all **preserve structure**

Type Refinement Systems



Type system: *Opfibration* of derivations and terms into the semantical category

Type Refinement Systems

Type Refinements: framing device so that every type system is a refinement system of some types

Curry-Howard-Lambek Correspondence:

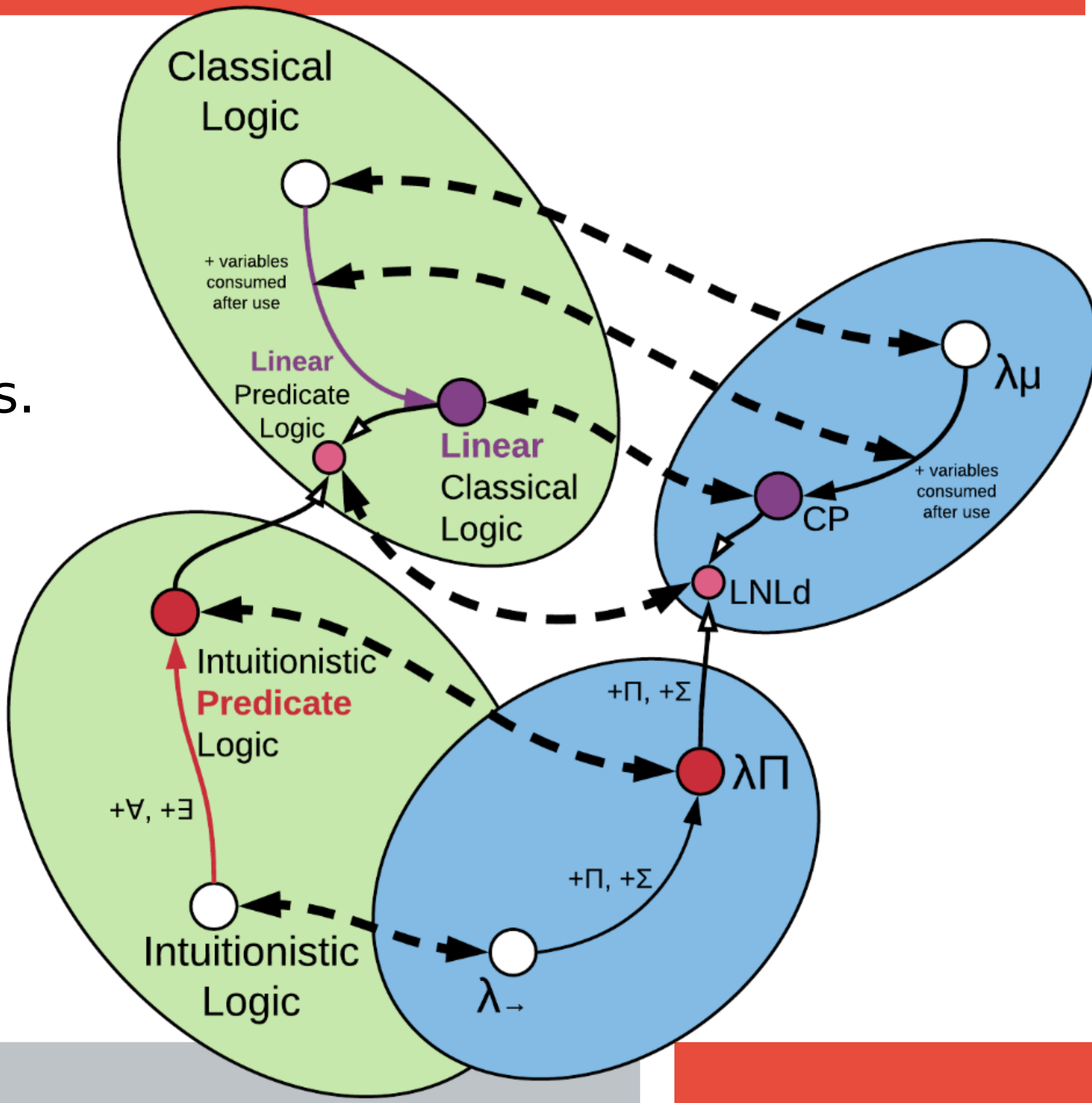
Logic	Calculus	Categorical Semantics
Intuitionistic	$\lambda_\to$	Cartesian Closed Categories
Classical	$\lambda\mu$	Topos in Cartesian Closed Categories
Intuitionistic Predicate	$\lambda\Pi$	Indexed Slice Category Fibrations
Linear Classical	CP	$*$-Autonomous Categories

Additionally, functors between categories can establish refinements.

The Crazy Part

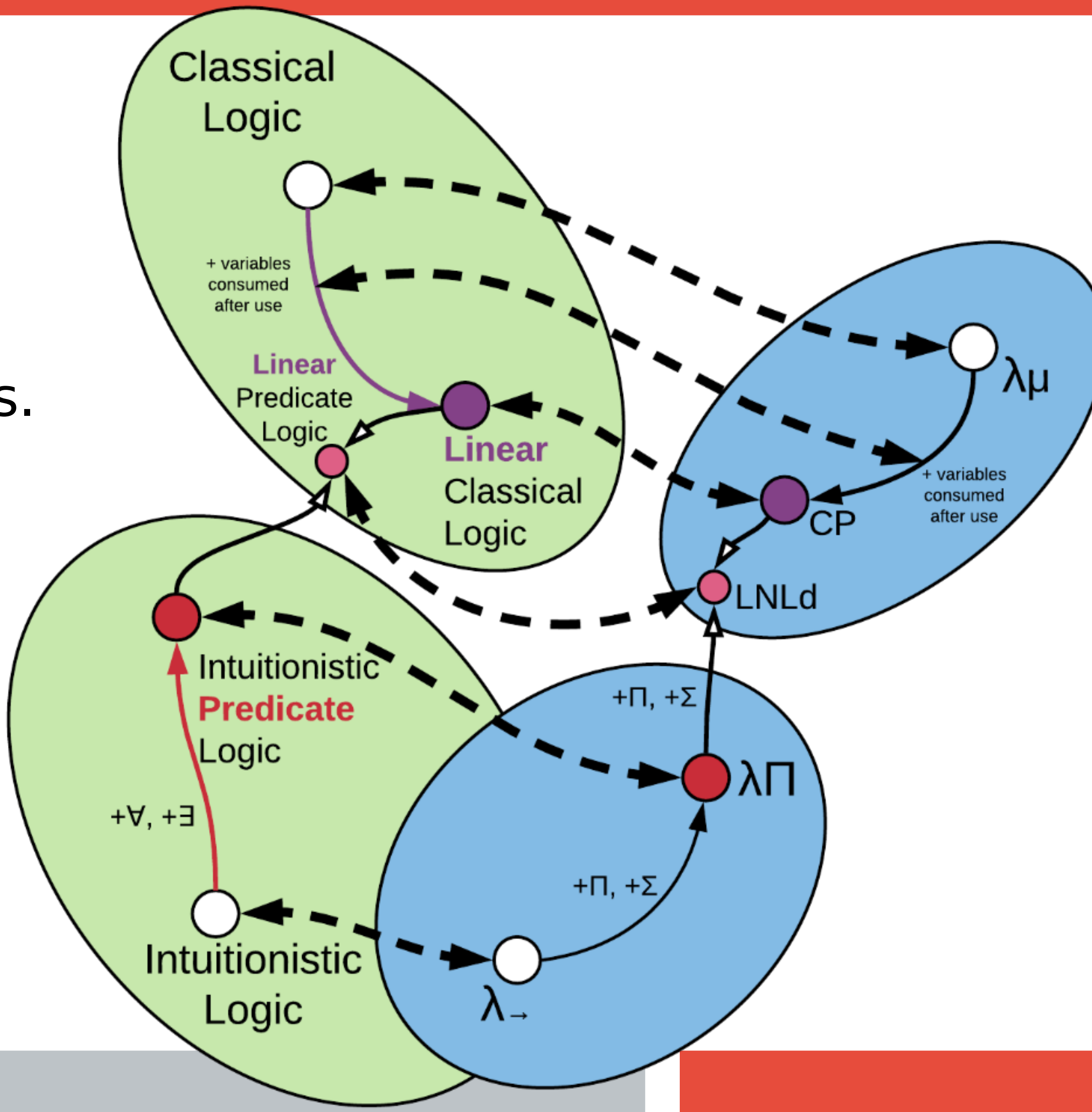
Curry-Howard seems to be a **functor** between logics (derivations), languages (terms), and categorical semantics.

Is there a type system on the level of the CHL correspondence?



The Crazy Part

Curry-Howard seems
to be a **functor** between
logics (derivations),
languages (terms),
and categorical semantics.



Is there a type system
on the level of the CHL
correspondence?

Is it the missing body
piece of this butterfly?

Outline

Motivation

Several interesting findings

A more refined perspective

Conclusion

In Review

Curry-Howard Correspondence

Makes our type systems more systematic

Gives proofs for “free” (e.g. deadlock in CP calculus)

Can support program behaviours if type system is built with relevant underlying logic that expresses the bounds for such behaviours

Is everywhere.

Opportunities

Spectrum of logics corresponding to automatic memory management methods

Exploring the space between known parts

Program synthesis via Curry-Howard

Solvers for category-theorical proofs

Opportunities

Spectrum of logics corresponding to automatic memory management methods

if reference counting is linear logic[21], what's garbage collection?

Exploring the space between known parts

Program synthesis via Curry-Howard

Solvers for category-theorical proofs

[21] Chirimar, Jawahar, Carl A. Gunter, and Jon G. Riecke. "Reference counting as a computational interpretation of linear logic." Journal of Functional Programming 6.2 (1996): 195-244.

Opportunities

Spectrum of logics corresponding to automatic memory management methods

Exploring the space between known parts

filling the gaps on the bridge
between logics and type systems

Program synthesis via Curry-Howard

Solvers for category-theorical proofs

Opportunities

Spectrum of logics corresponding to automatic memory management methods

Exploring the space between known parts

Program synthesis via Curry-Howard

Solvers for category-theorical proofs

Opportunities

Spectrum of logics corresponding to automatic memory management methods

Exploring the space between known parts

Program synthesis via Curry-Howard

Solvers for category-theorical proofs

Thank You