Comparing functional programs, or how to put λ-terms back in order

Adrienne Lancelot
Thesis prepared at the *Institut Polytechnique de Paris*

Advisors: Beniamino Accattoli and Giulio Manzonetto
Inria & LIX, École Polytechnique and Université Paris Cité, CNRS, IRIF

E.W. Beth Dissertation Award, August 7th 2026

Theoretical Computer Science → Programming Languages

Comparing functional programs, or how to put λ-terms back in order

Studying functional programming languages

```
let sort (compare :  'a -> 'a -> bool) :  list 'a -> list 'a := ...
```

Investigating the problem of program equivalence

Are `quick_sort` and `bubble_sort` equivalent?

In Practice, program equivalence is used to verify the correctness of the changes made by the user or the compiler.

Theoretical Computer Science → Programming Languages

Comparing **functional programs**, or how to put **λ-terms** back in order

Studying functional programming languages

```
let sort (compare :  'a -> 'a -> bool) :  list 'a -> list 'a := ...
```

Investigating the problem of program equivalence

Are `quick_sort` and `bubble_sort` equivalent?

In Practice, program equivalence is used to verify the correctness of the changes made by the user or the compiler.

Theoretical Computer Science → Programming Languages

Comparing functional programs, or how to put λ-terms back in order

Studying functional programming languages

```
let sort (compare :  'a -> 'a -> bool) :  list 'a -> list 'a := ...
```

Investigating the problem of program equivalence

Are `quick_sort` and `bubble_sort` equivalent?

In Practice, program equivalence is used to verify the correctness of the changes made by the user or the compiler.

Theoretical Computer Science → Programming Languages

Comparing functional programs, or how to put λ-terms back in order

Studying functional programming languages

```
let sort (compare :  'a -> 'a -> bool) :  list 'a -> list 'a := ...
```

Investigating the problem of program equivalence

Are `quick_sort` and `bubble_sort` equivalent?

In Practice, program equivalence is used to verify the correctness of the changes made by the user or the compiler.

Comparing functional programs, or how to put λ-terms back in **order**

Studying functional programming languages

```
let sort (compare :  'a -> 'a -> bool) :  list 'a -> list 'a := ...
```

Investigating the problem of program equivalence

Are `quick_sort` and `bubble_sort` equivalent?

In Practice, program equivalence is used to verify the correctness of the changes made by the user or the compiler.

Order(s) on λ-terms

How (in what **order**) does the computer compute syntactic programs?
$\rightarrow$ computation dynamics: operational semantics, evaluation strategies, rewriting

Program **Preorders** instead of Equivalences!
$\rightarrow$ Updating code instead of always writing equivalent programs

Improvement Theory: computation **costs** of programs
$\rightarrow$ Quick sort is not only equivalent, but *better* than bubble sort

Order(s) on λ-terms

How (in what **order**) does the computer compute syntactic programs?
$\rightarrow$ computation dynamics: operational semantics, evaluation strategies, rewriting

Program **Preorders** instead of Equivalences!
$\rightarrow$ Updating code instead of always writing equivalent programs

Improvement Theory: computation **costs** of programs
$\rightarrow$ Quick sort is not only equivalent, but *better* than bubble sort

Order(s) on λ-terms

How (in what **order**) does the computer compute syntactic programs?
$\rightarrow$ computation dynamics: operational semantics, evaluation strategies, rewriting

Program **Preorders** instead of Equivalences!
$\rightarrow$ Updating code instead of always writing equivalent programs

Improvement Theory: computation **costs** of programs
$\rightarrow$ Quick sort is not only equivalent, but *better* than bubble sort

Evaluation Order

My program

```
let double (x :  int) :  int := x + x.
let my_computation := double(2+3).
```

Evaluation Order

My program

```
let double (x :  int) :  int := x + x.
let my_computation := double(2+3).
```

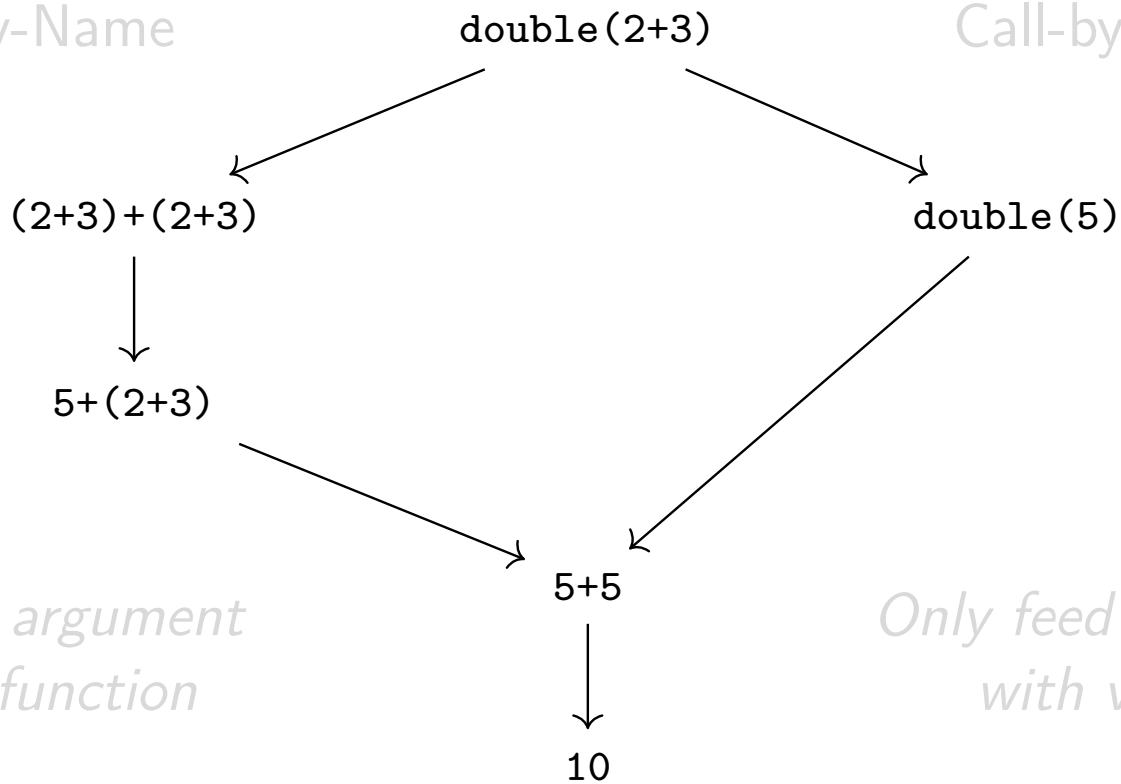
Evaluation Order

My program

```
let double (x :  int) :  int := x + x.
let my_computation := double(2+3).
```

Call-by-Name

Call-by-Value



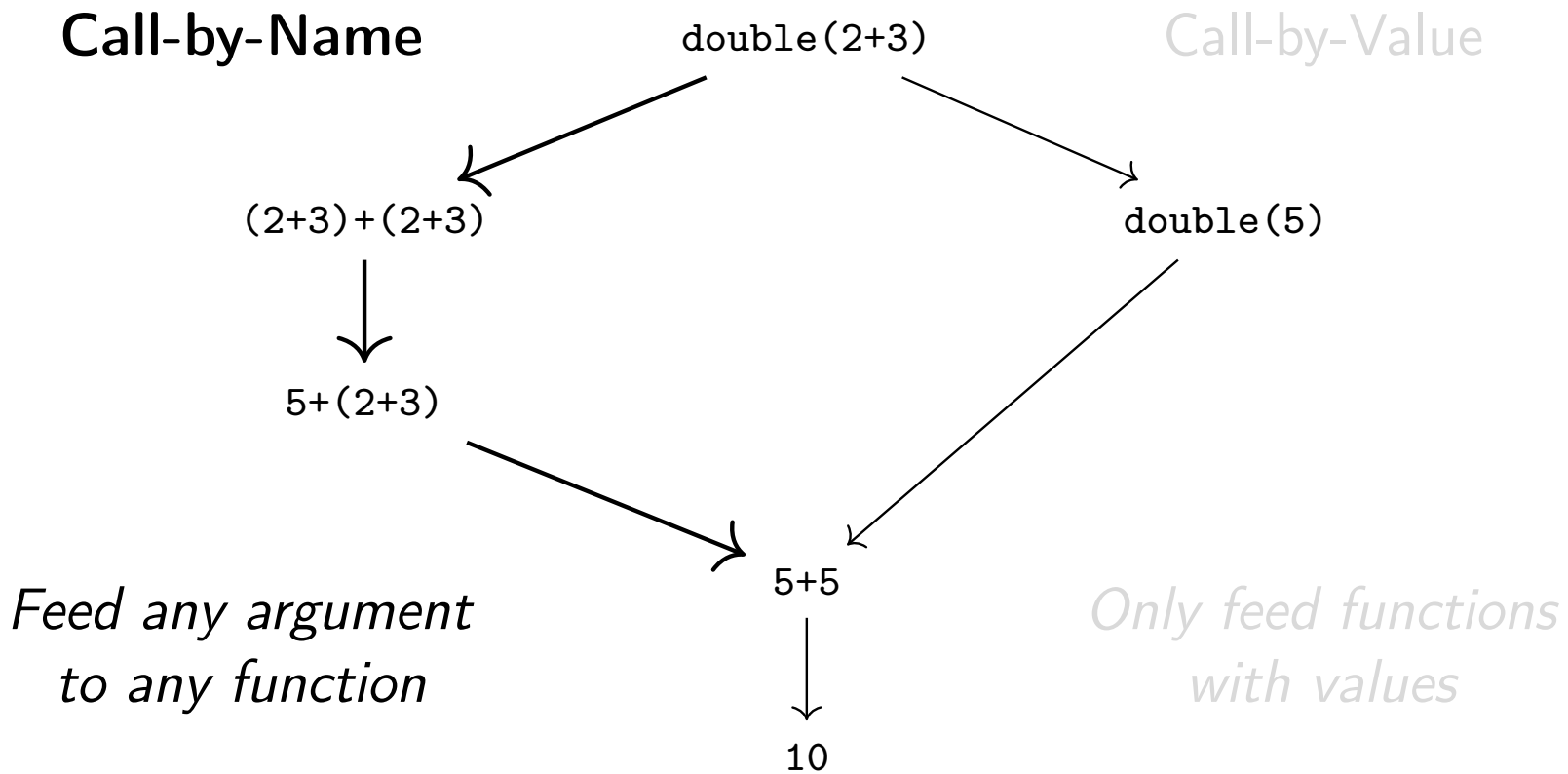
Feed any argument to any function

Only feed functions with values

Evaluation Order

My program

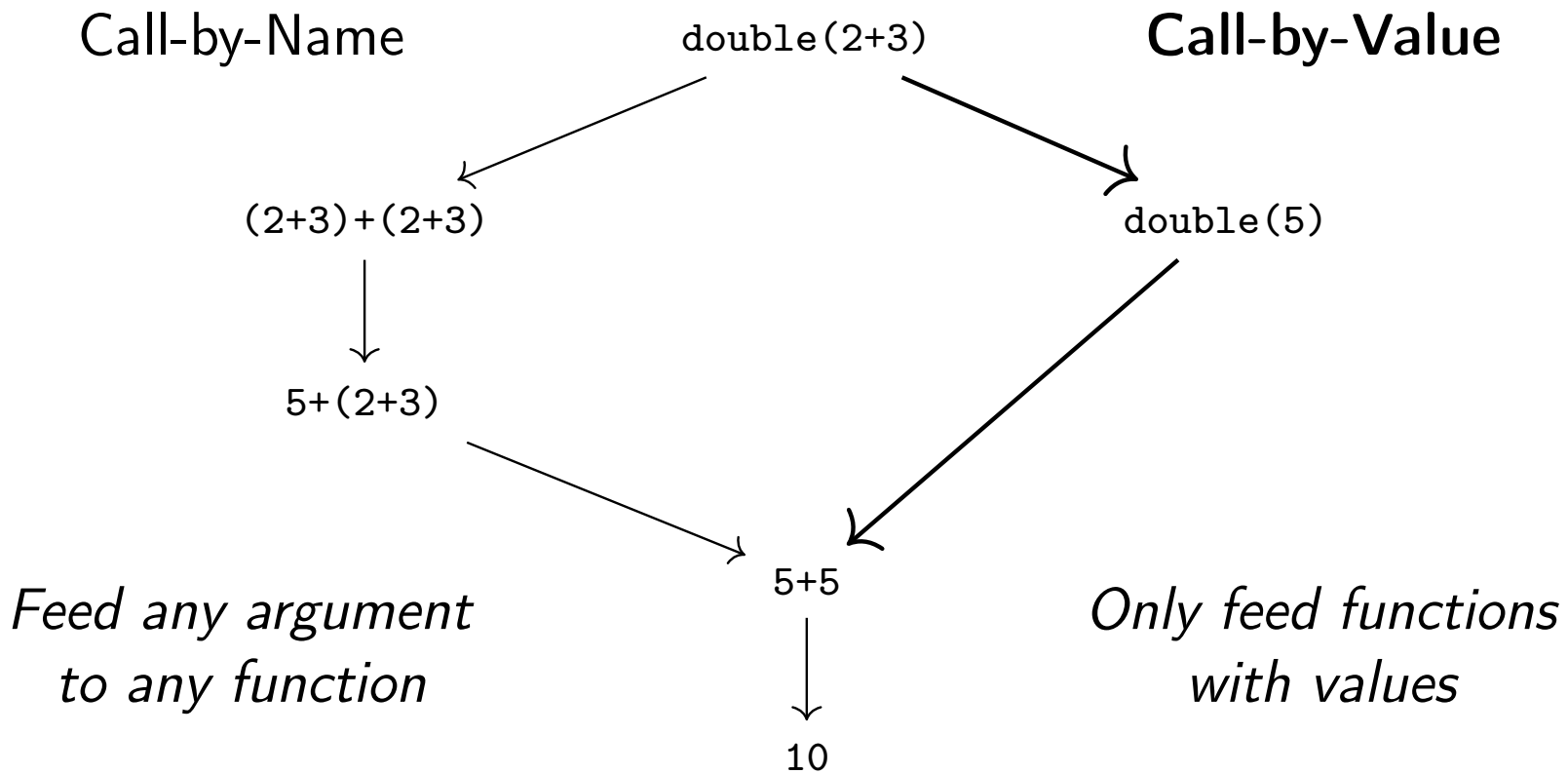
```
let double (x :  int) :  int := x + x.
let my_computation := double(2+3).
```



Evaluation Order

My program

```
let double (x :  int) :  int := x + x.
let my_computation := double(2+3).
```



Computational Equivalence

Two coders write a program, how do we know if they are the same?

- **Inspecting the Code:**

A program

```
let my_function :=
```

Another program

```
let my_function :=
```

Computational Equivalence

Two coders write a program, how do we know if they are the same?

- **Inspecting the Code:**

A program

```
let my_function :=
            ...
```

Another program

```
let my_function :=
            ...
```

Computational Equivalence

Two coders write a program, how do we know if they are the same?

- **Inspecting the Code:**

A program

```
let my_function :=
                2+3.
```

Another program

```
let my_function :=
                2+3.
```

The code is identical.

Computational Equivalence

Two coders write a program, how do we know if they are the same?

- **Inspecting the Code:**

A program

```
let my_function :=
          2+3+2+3.
```

Another program

```
let my_function :=
      double(2+3).
```

The code is computationally equivalent.

Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

```
let n := 42.  ...
```

A program

```
let my_fun x
:=
              x+x.
```

The same execution environment

```
let n := 42.  ...
```

Another program

```
let my_fun x
:=
              2*x.
```

Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

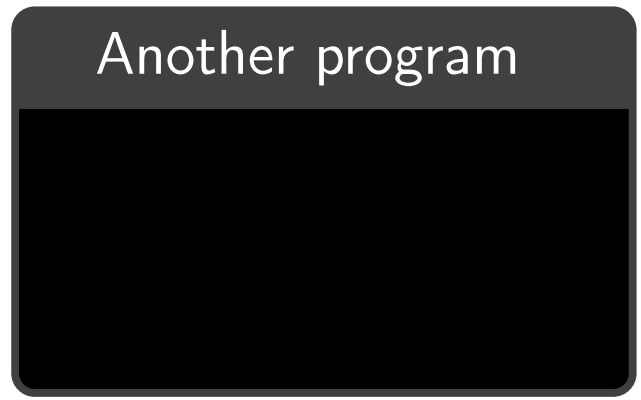
```
let n := 42.  ...
```

A program

The same execution environment

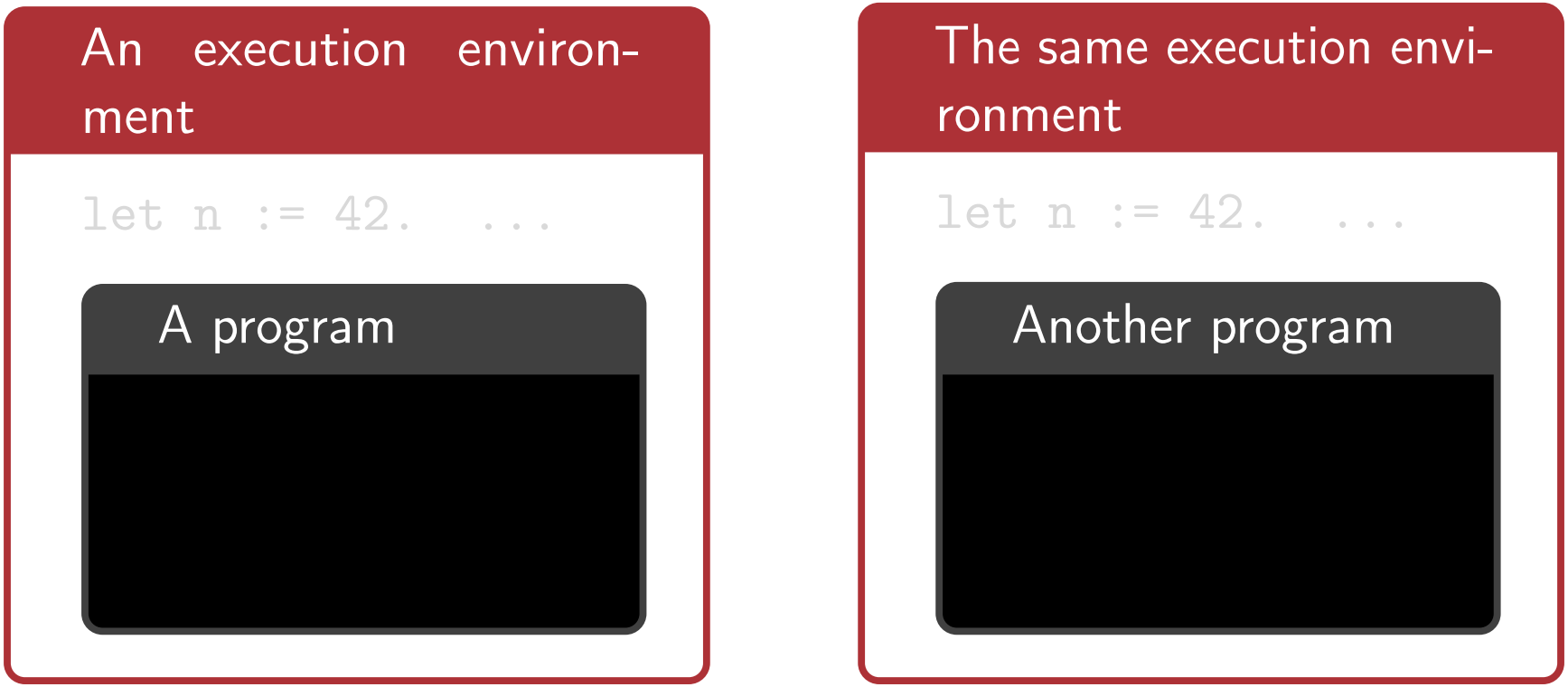
```
let n := 42.  ...
```

Another program



Contextual Equivalence, Morris [1968]

- **Using the Programs:**



The code is tested.

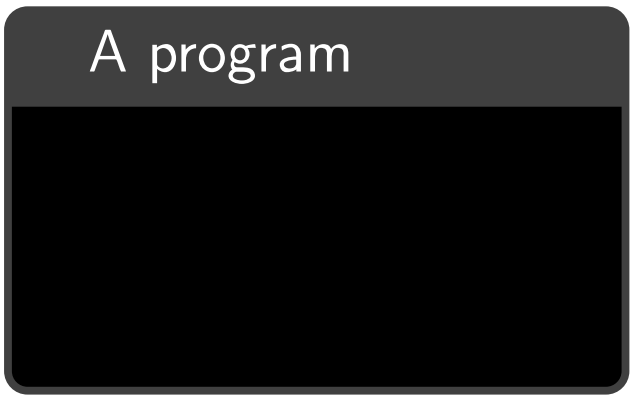
Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

```
let n := 42.  ...
```

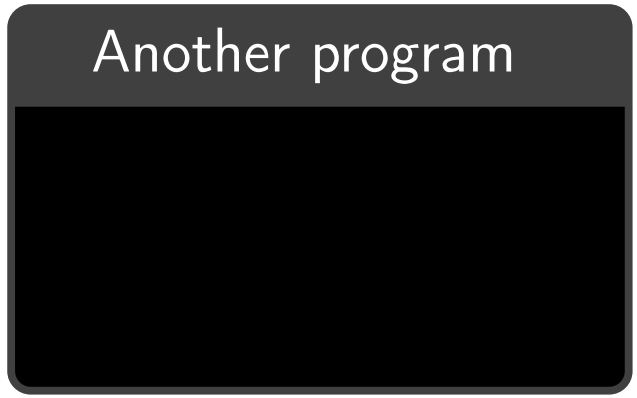
A program



The same execution environment

```
let n := 42.  ...
```

Another program



The context affects the behavior of the program.

Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

```
let n := 42.  ...
```

A program

```
n
```

The same execution environment

```
let n := 42.  ...
```

Another program

```
n
```

The context affects the behavior of the program.

Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

```
let n := 42.  ...
```

A program

```
my_fun
```

The same execution environment

```
let n := 42.  ...
```

Another program

```
my_fun
```

The context affects the behavior of the program.

Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

```
let n := 42.   ...
```

A program

```
my_fun
```

```
...
print(my_fun(n));
```

The same execution environment

```
let n := 42.   ...
```

Another program

```
my_fun
```

```
...
print(my_fun(n));
```

The context affects the behavior of the program.

Contextual Equivalence, Morris [1968]

- **Using the Programs:**

An execution environment

```
let n := 42.   ...
```

A program

```
my_fun
```

```
...
print(my_fun(n));
```

```
(* terminates, prints 84 *)
```

The same execution environment

```
let n := 42.   ...
```

Another program

```
my_fun
```

```
...
print(my_fun(n));
```

```
(* terminates, prints 84 *)
```

The context affects the behavior of the program.

Contextual Equivalence, Morris [1968]

- **Using the Programs:** $\forall$ execution environment

An execution environment

```
let n := 42.
...
```

```
...
print(my_fun(n));
```

The same execution environment

```
let n := 42.
...
```

```
...
print(my_fun(n));
```

Testing programs in every possible context.

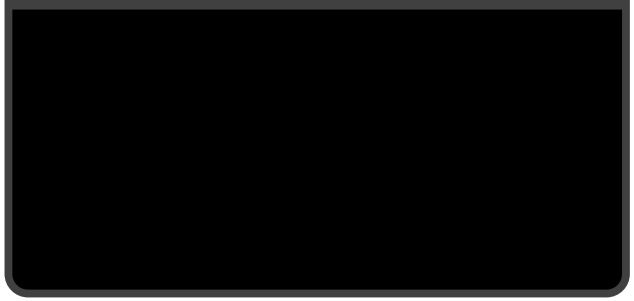
Contextual Equivalence, Morris [1968]

- **Using the Programs:** $\forall$ execution environment

An execution environment

```
let n := 42.  ...
```

A program



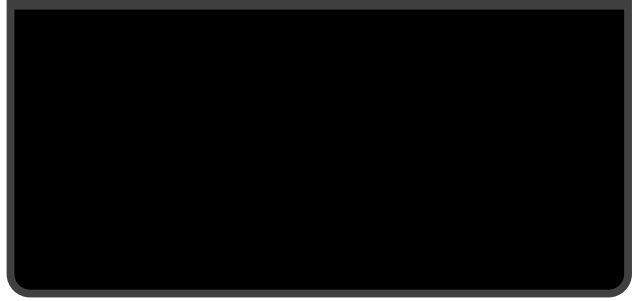
```
...
print(my_fun(n));
```

```
(* terminates, prints x *)
```

The same execution environment

```
let n := 42.  ...
```

Another program



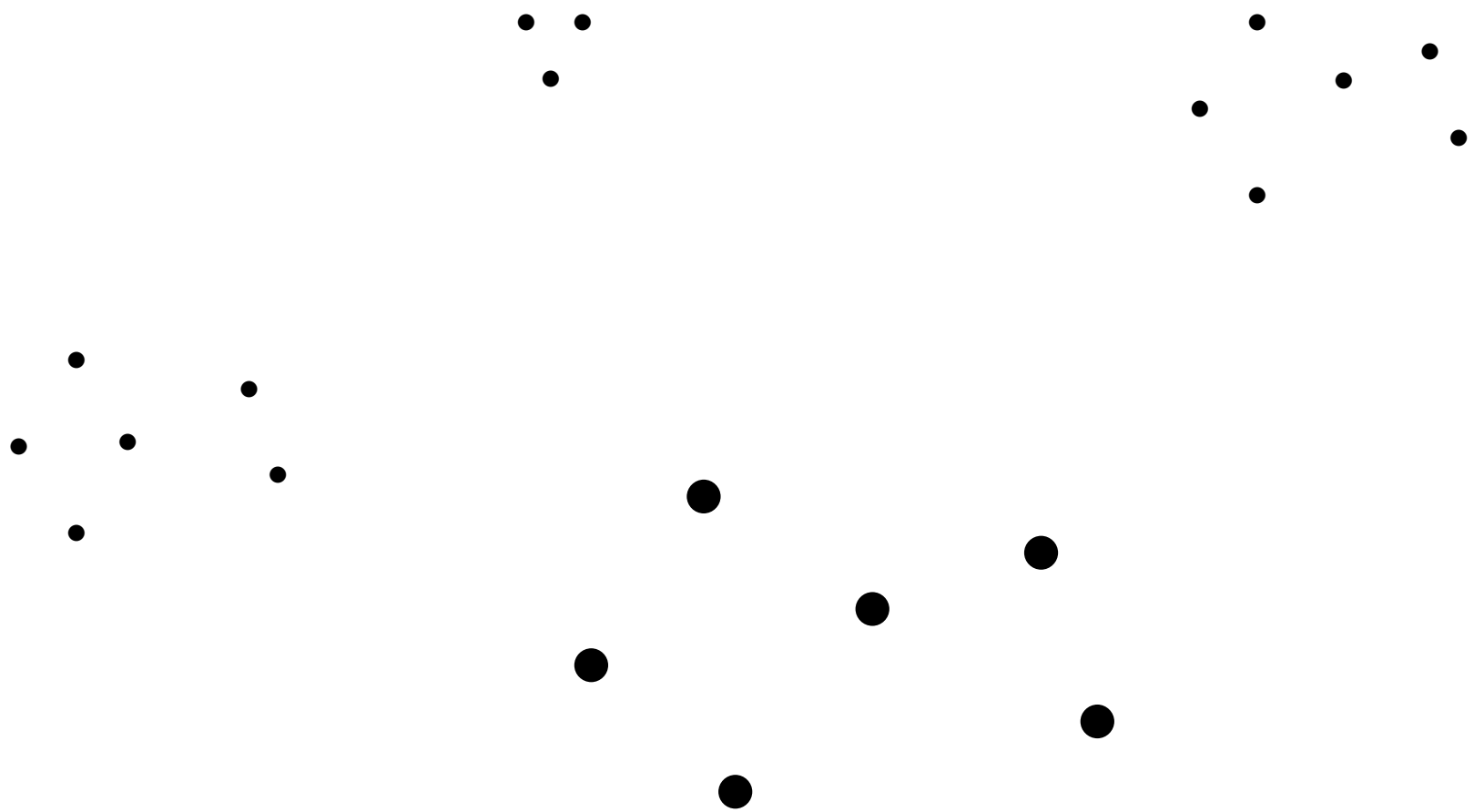
```
...
print(my_fun(n));
```

```
(* terminates, prints x *)
```

Testing programs in every possible context.

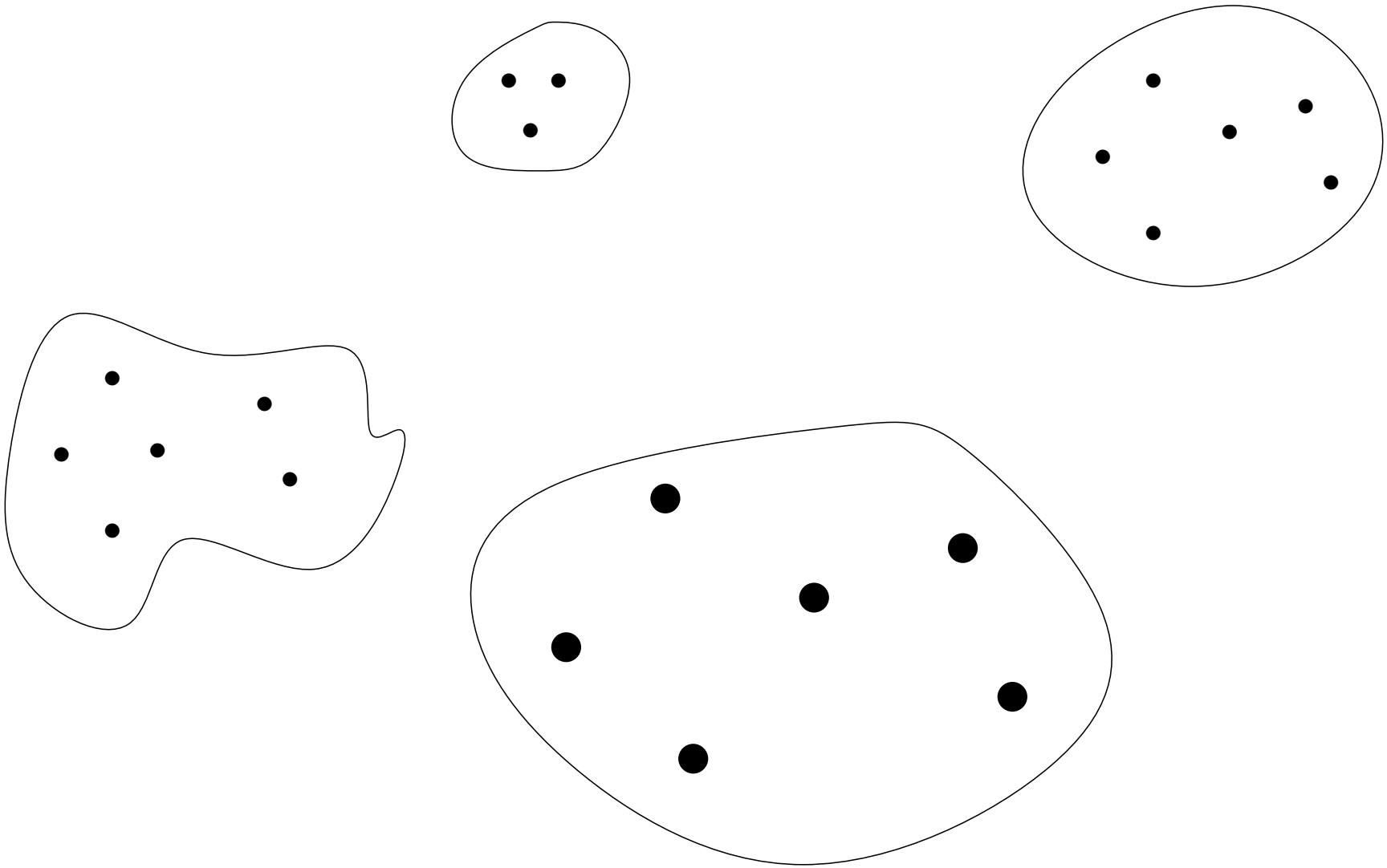
(In)Equational Theories

Program Equivalences in General:



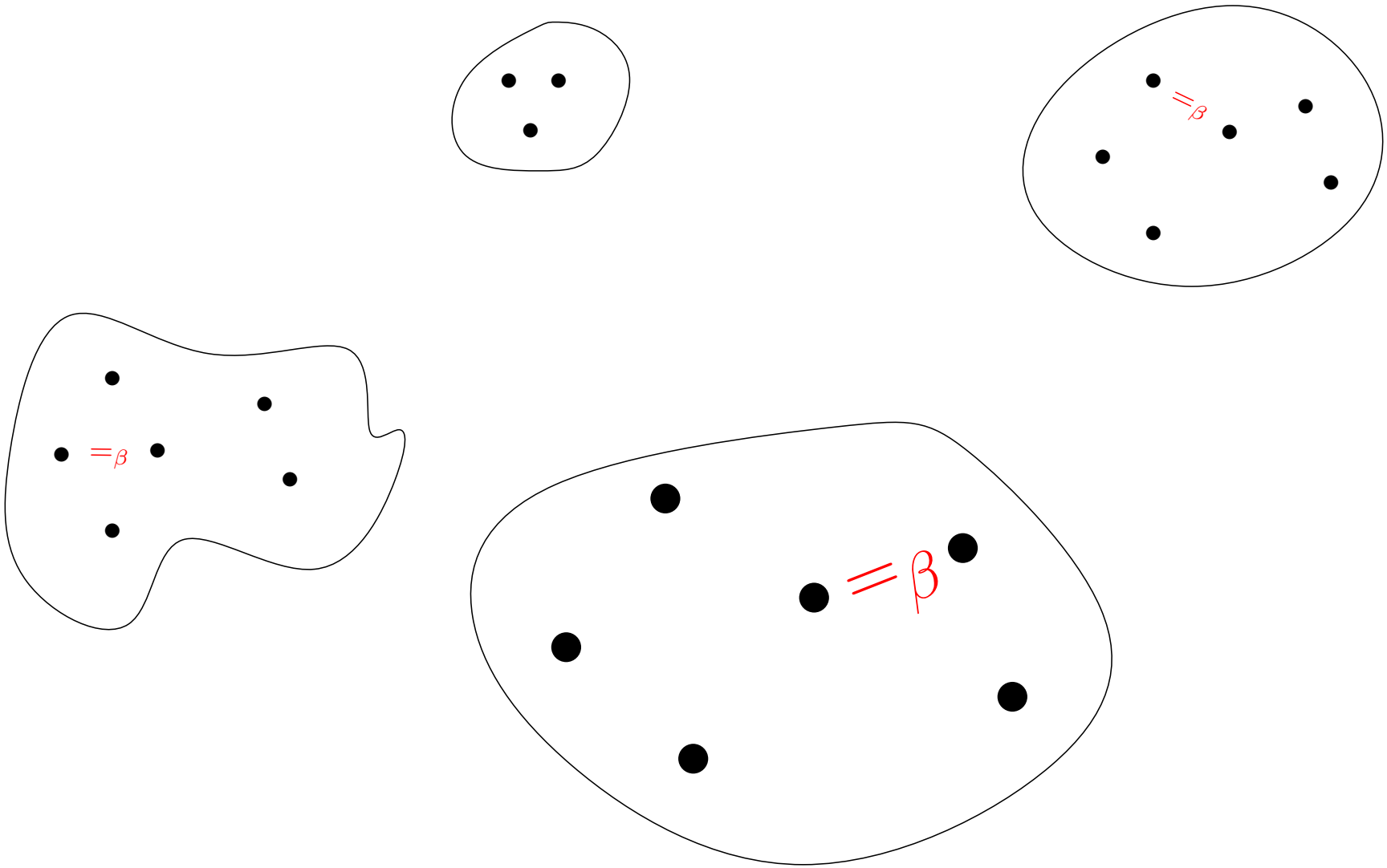
(In)Equational Theories

Program Equivalences in General:



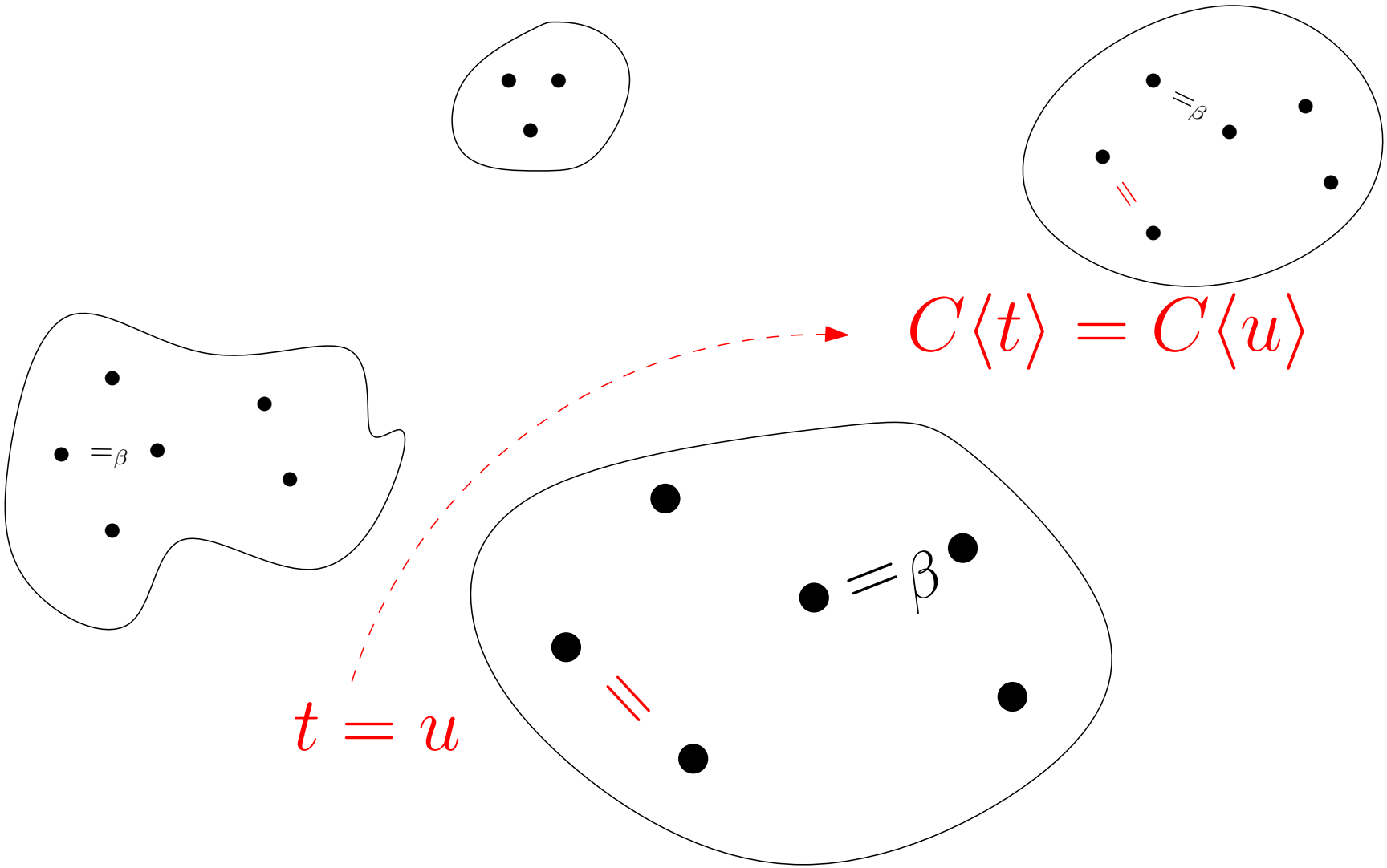
(In)Equational Theories

Program Equivalences in General:



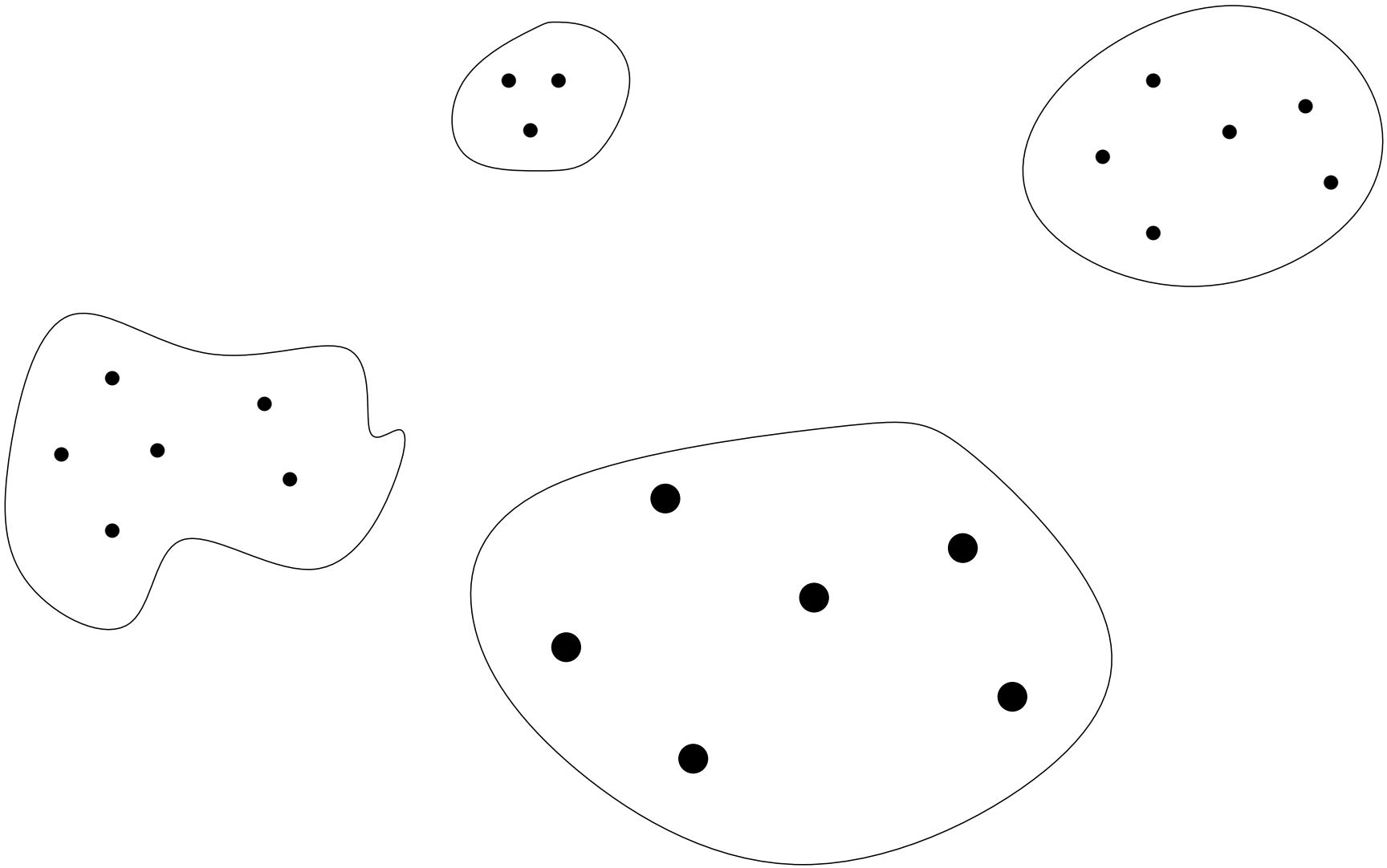
(In)Equational Theories

Program Equivalences in General:



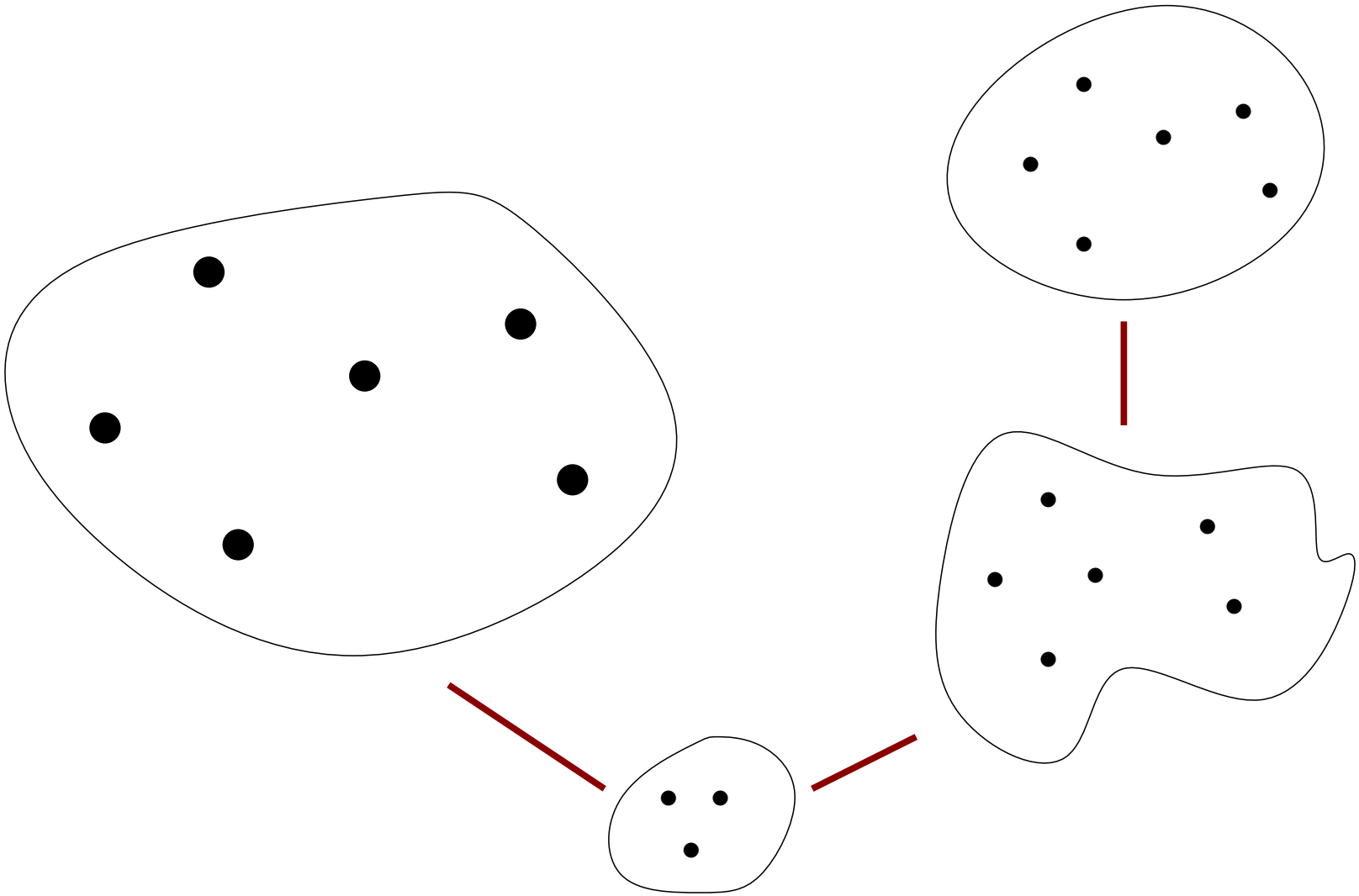
(In)Equational Theories

Program ~~Equivalences~~ in General:



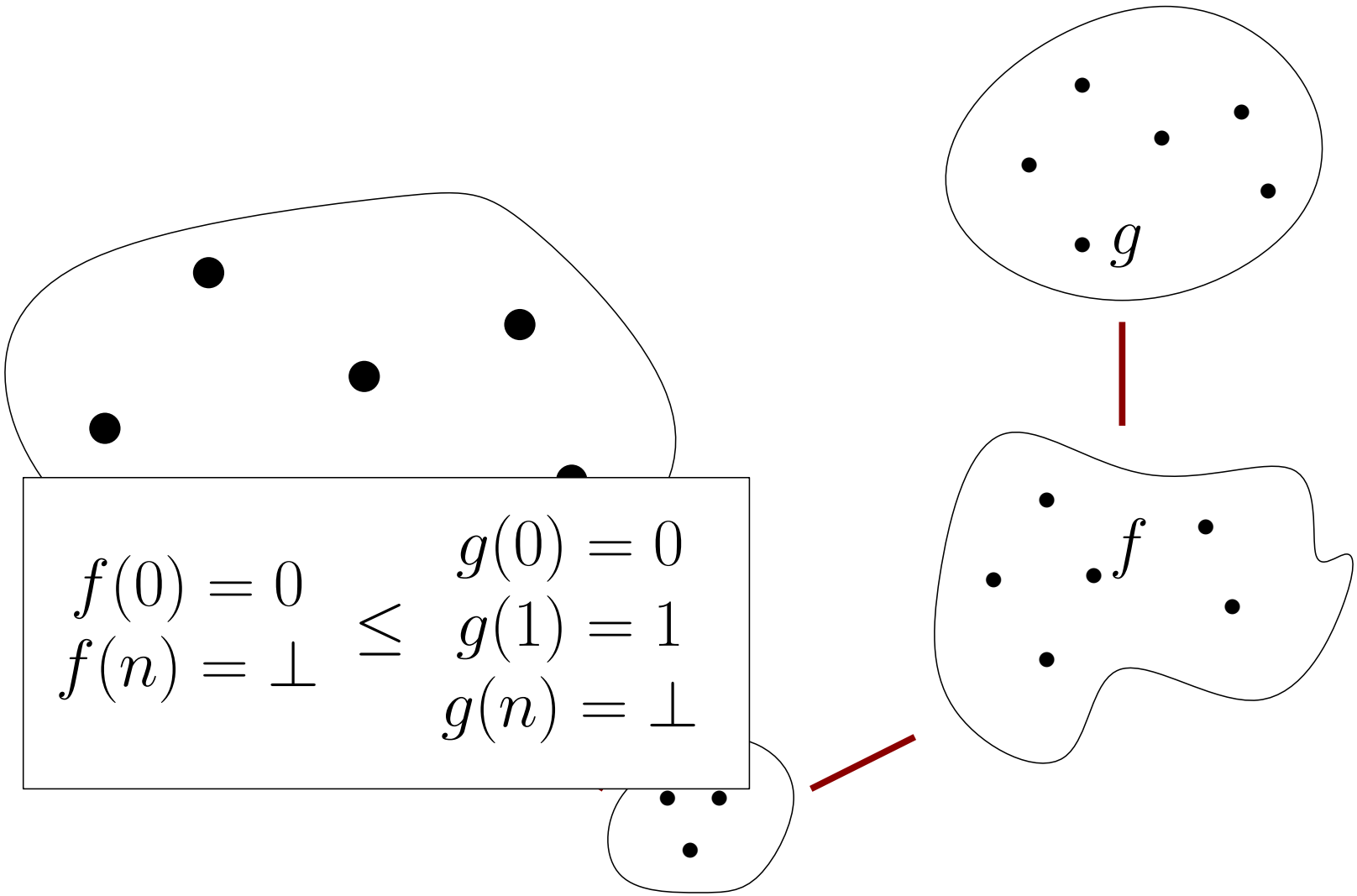
(In)Equational Theories

Program ~~Equivalences~~ Preorders in General:



(In)Equational Theories

Program ~~Equivalences~~ Preorders in General:



Which (In)Equational Theory, and for what?

- Contextual Equivalence

  a natural equational theory

- Tractable Program Equivalences

  e.g. bisimulations

- Cost-Aware Program Equivalences

  $\rightarrow$ intensional properties of programs

Which (In)Equational Theory, and for what?

- Contextual Equivalence

  a natural equational theory

- Tractable Program Equivalences

  e.g. bisimulations

- Cost-Aware Program Equivalences

  $\rightarrow$ intensional properties of programs

Which (In)Equational Theory, and for what?

- Contextual Equivalence

  a natural equational theory

- Tractable Program Equivalences

  e.g. bisimulations

- Cost-Aware Program Equivalences

  $\rightarrow$ intensional properties of programs

Interaction Equivalence

> « *The meaning of a program should express its history of access to resources which are not local to it.* »
>
> — Robin Milner [1975]

Refine **quantitatively** contextual equivalence, ignoring the internal behavior of programs!

Interaction Equivalence

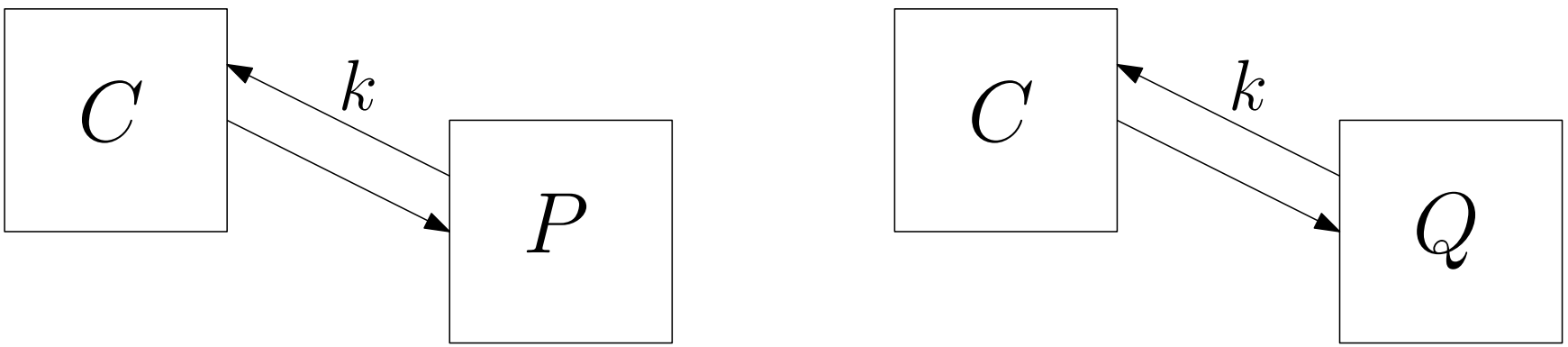
> « *The meaning of a program should express its history of access to resources which are not local to it.* »
>
> — Robin Milner [1975]

Refine **quantitatively** contextual equivalence, ignoring the internal behavior of programs!

Interaction Equivalence

> « *The meaning of a program should express its history of access to resources which are not local to it.* »
>
> — Robin Milner [1975]

Refine **quantitatively** contextual equivalence, ignoring the internal behavior of programs!

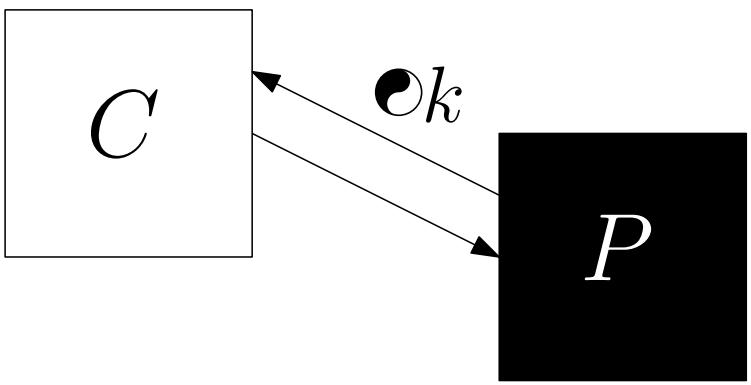


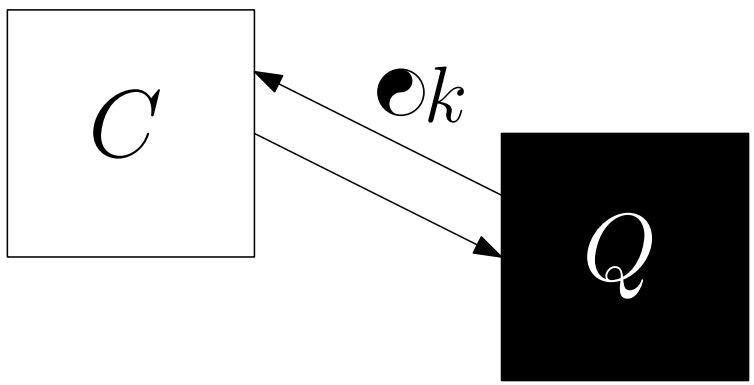
$$P \equiv^{\text{int}} Q$$

Interaction Equivalence

> « *The meaning of a program should express its history of access to resources which are not local to it.* »
>
> — Robin Milner [1975]

Refine **quantitatively** contextual equivalence, ignoring the internal behavior of programs!

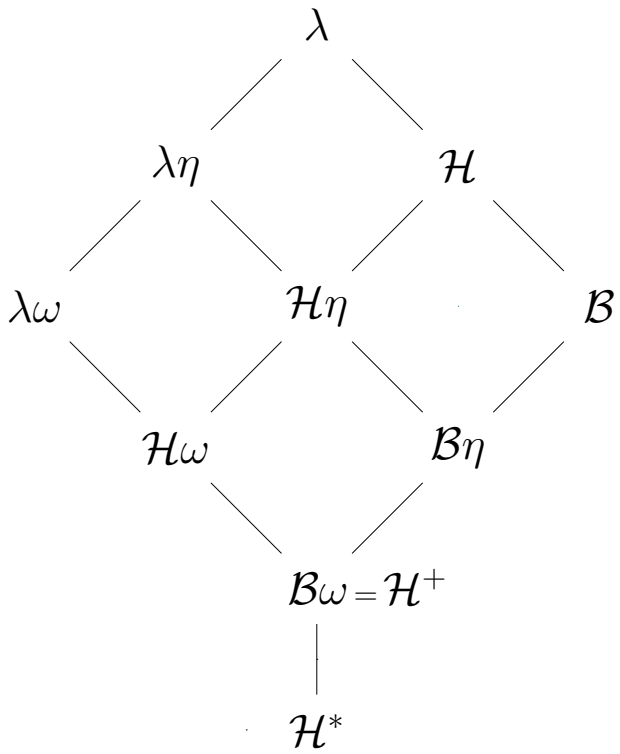




$$(\lambda_\bullet x.t) \bullet u \to_\tau t\{x \leftarrow u\}$$

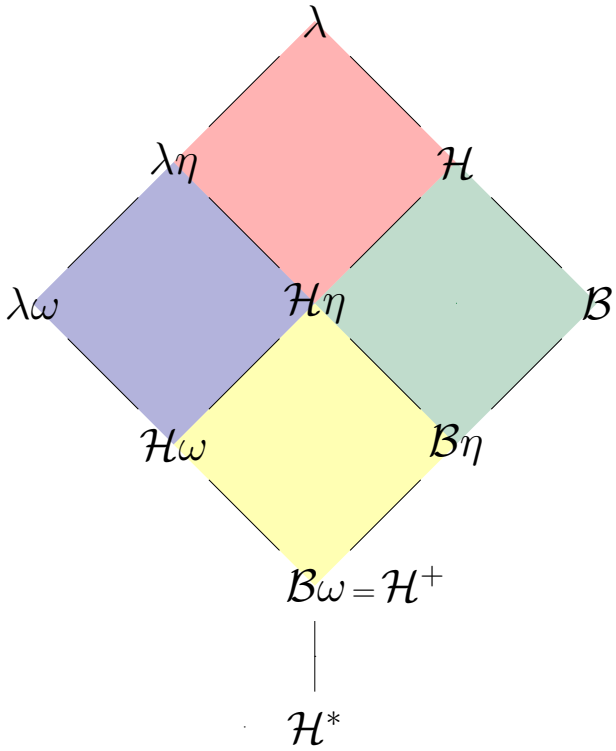
$$(\lambda_\bullet x.t) \circ u \to_\circ t\{x \leftarrow u\} \qquad P \equiv^{\text{int}} Q$$

A Call-by-Value Odyssey



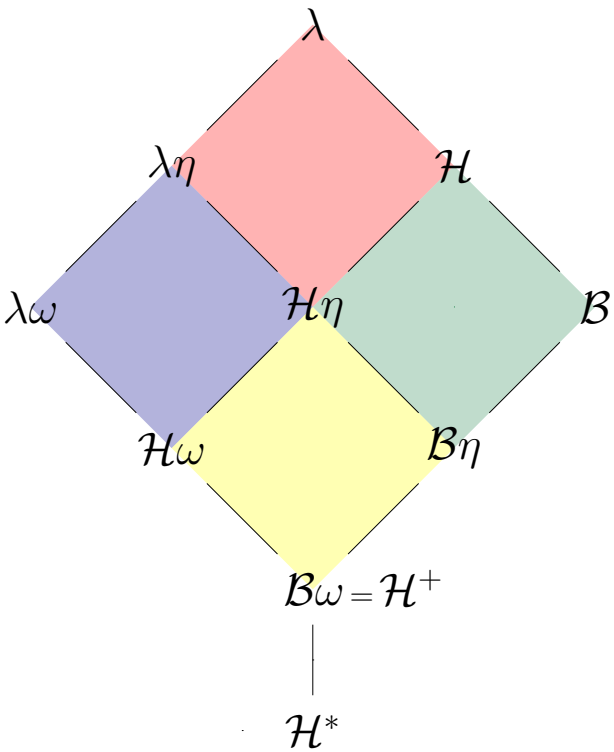
$(x + y) + (x + y) \quad = \quad \texttt{let } z = (x + y) \texttt{ in } z + z$

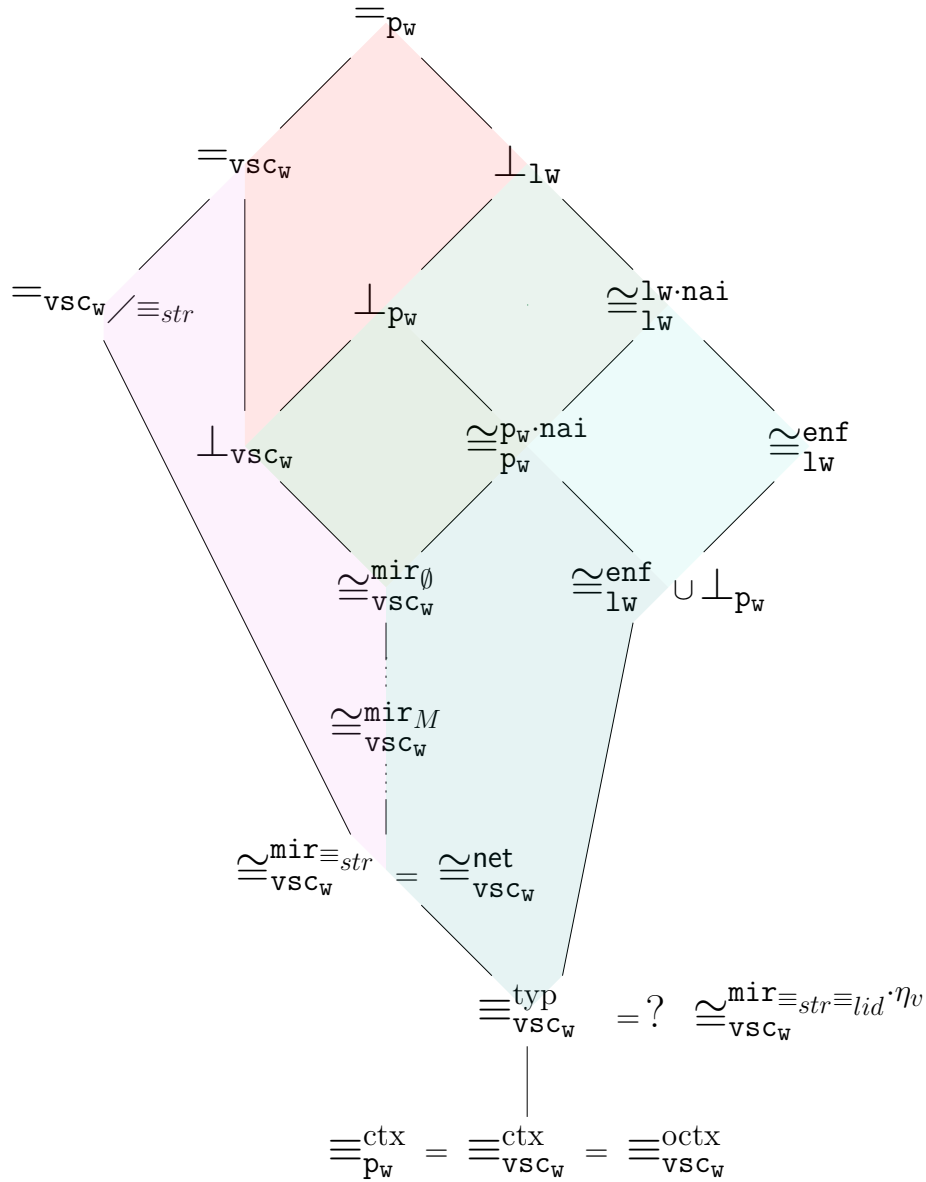
A Call-by-Value Odyssey



$(x + y) + (x + y) \quad = \quad \texttt{let } z = (x + y) \texttt{ in } z + z$

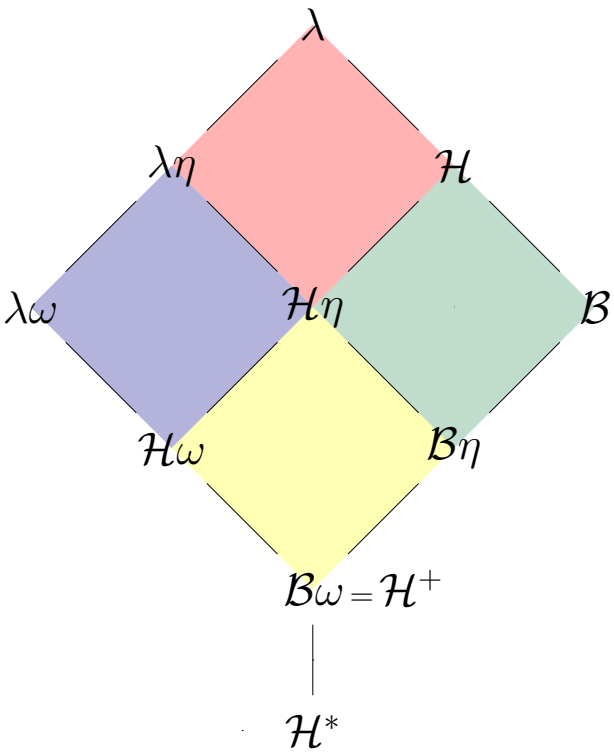
A Call-by-Value Odyssey

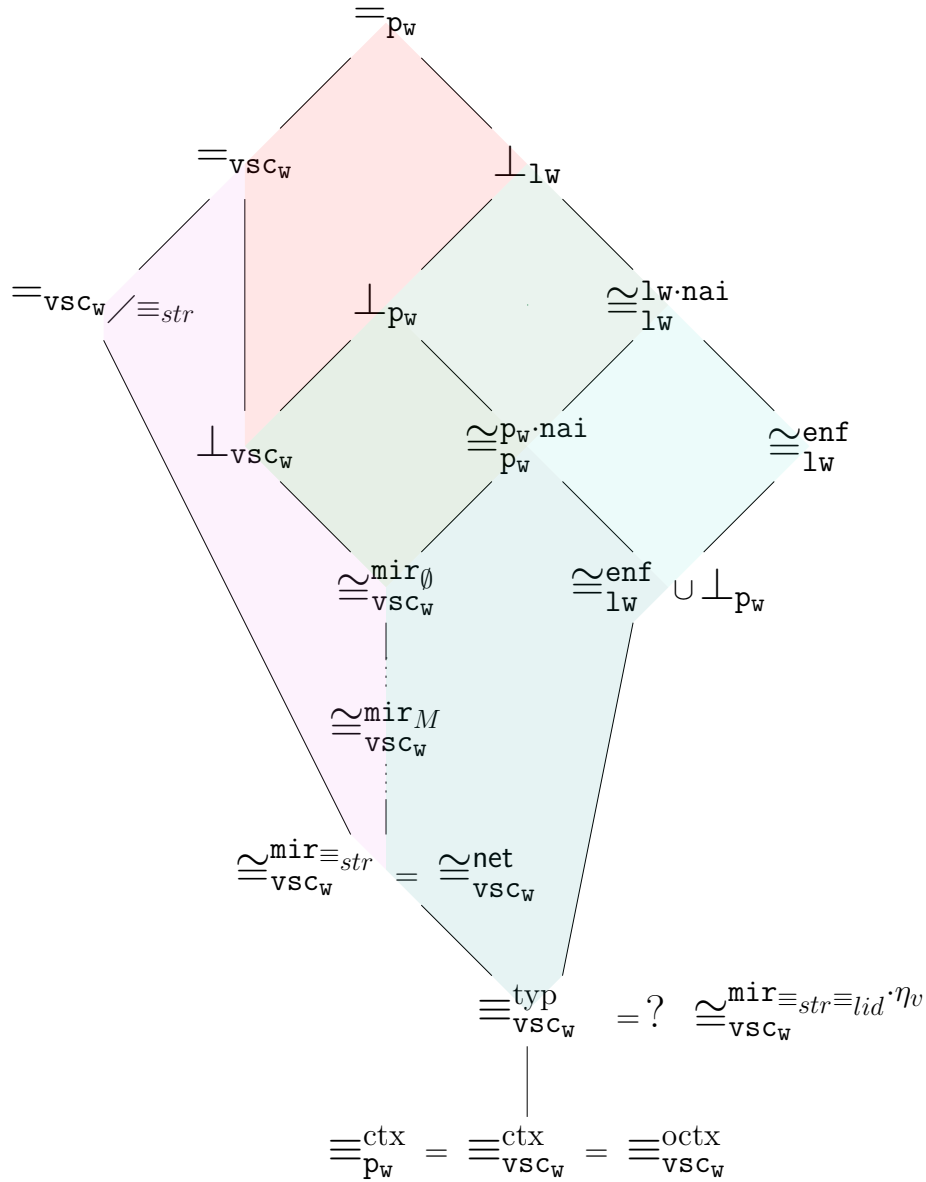




$(\mathtt{x}+\mathtt{y})+(\mathtt{x}+\mathtt{y}) \quad \equiv \quad \mathtt{let\ z} = (\mathtt{x}+\mathtt{y})\ \mathtt{in\ z}+\mathtt{z}$

A Call-by-Value Odyssey





$$(\mathtt{x}+\mathtt{y})+(\mathtt{x}+\mathtt{y}) \quad \equiv \quad \mathtt{let\ z} = (\mathtt{x}+\mathtt{y})\ \mathtt{in\ z}+\mathtt{z}$$

What's Next?

- Now: Postdoc at Università di Bologna, Italy
- **Interactive Semantics**

  composition = synchronization = substitution
- A Theory of Costs for Compilers?

What's Next?

- Now: Postdoc at Università di Bologna, Italy
- **Interactive Semantics**

  composition = synchronization = substitution
- A Theory of Costs for Compilers?

What's Next?

- Now: Postdoc at Università di Bologna, Italy
- **Interactive Semantics**

  composition = synchronization = substitution
- A Theory of Costs for Compilers?

« Because of its precisely formulated mathematical properties, the λ-calculus transcends subjective debate. By defining its semantics in the same way as for any other programming language, there is an opportunity (unique?) to scrutinize our theories. »

— Christopher P. Wadsworth [1971]

Thank you!

« Because of its precisely formulated mathematical properties, the λ-calculus transcends subjective debate. By defining its semantics in the same way as for any other programming language, there is an opportunity (unique?) to scrutinize our theories. »

— Christopher P. Wadsworth [1971]

Thank you!

Theory of Computation

Church's λ-Calculus

- A prototypical functional programming language

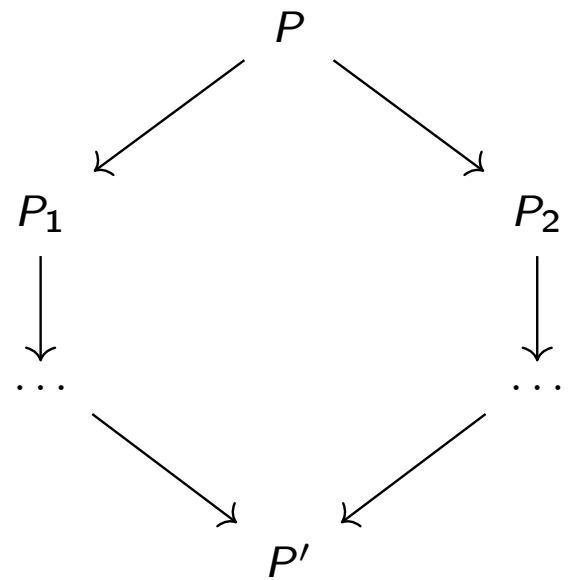
$x \mapsto f(x)$ is $\lambda x.f(x)$

β-reduction:

$(\lambda x.x+x)(n) \to_\beta n+n$

Rewriting Theory

- Study of Rewriting Sequences

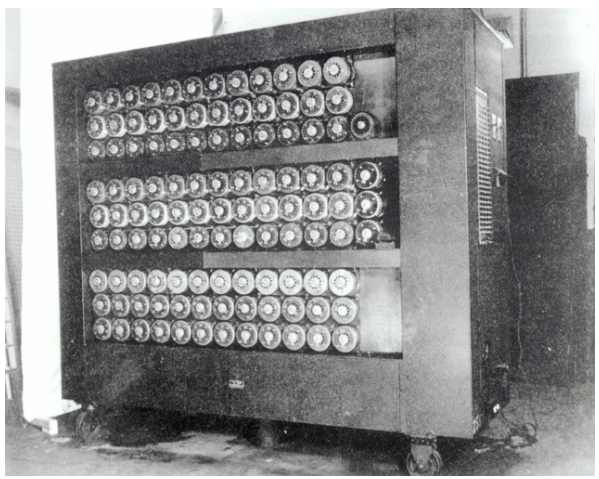


Semantics as Specification

Program equivalence can be studied through different **semantic** point of views.

Operational **Denotational**

Machine $\xleftarrow[\textit{interpreter}]{}$ Language $\xrightarrow{\textit{interpretation}}$ Abstract Math. Structures



$$[\![\texttt{myprog}]\!] = f : \mathbb{N} \to \mathbb{N}$$

$$[\![\texttt{myprog\_1(myprog\_2)}]\!]$$
$$\|$$
$$[\![\texttt{myprog\_1}]\!] \circ [\![\texttt{myprog\_2}]\!]$$

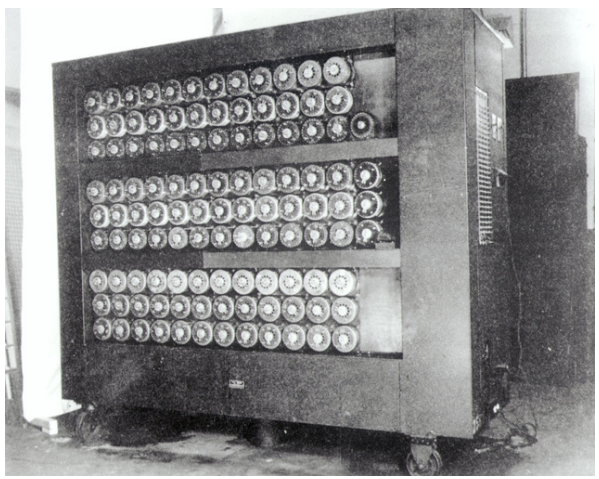
Non-idempotent intersection types can be seen as both operational and denotational semantics.

Semantics as Specification

Program equivalence can be studied through different **semantic** point of views.

Operational **Denotational**

Machine $\xleftarrow[\textit{interpreter}]{}$ Language $\xrightarrow{\textit{interpretation}}$ Abstract Math. Structures



$$[\![\texttt{myprog}]\!] = f : \mathbb{N} \to \mathbb{N}$$

$$[\![\texttt{myprog\_1(myprog\_2)}]\!]$$
$$\|$$
$$[\![\texttt{myprog\_1}]\!] \circ [\![\texttt{myprog\_2}]\!]$$

Non-idempotent intersection types can be seen as both operational and denotational semantics.

Contextual Equivalence depends on the Evaluation Paradigm

$$10 \xleftarrow{\text{CbN}} \texttt{double(2+3)} \xrightarrow{\text{CbV}} 10$$

There are looping terms like $\Omega \to \Omega \to \Omega \to \cdots$
Fix a program $\texttt{always\_1} := \lambda x.1$. How to compute $\texttt{always\_1}(\Omega)$?

Call-by-Name

Call-by-Value

Feed any argument to any function

Only feed functions with values

Contextual Equivalence depends on the Evaluation Paradigm

$$10 \xleftarrow{\text{CbN}} \texttt{double(2+3)} \xrightarrow{\text{CbV}} 10$$

There are **looping** terms like $\Omega \to \Omega \to \Omega \to \cdots$

Fix a program $\texttt{always\_1} := \lambda x.1$. How to compute $\texttt{always\_1}(\Omega)$?

Call-by-Name

Call-by-Value

Feed any argument to any function

Only feed functions with values

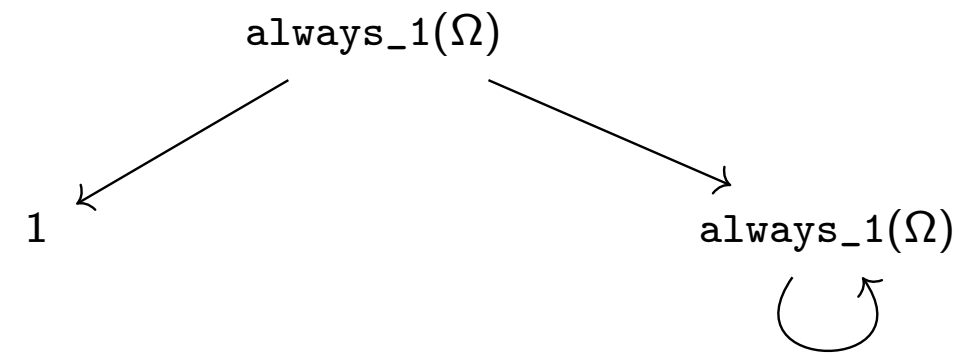
Contextual Equivalence depends on the Evaluation Paradigm

$$10 \xleftarrow{\text{CbN}} \texttt{double(2+3)} \xrightarrow{\text{CbV}} 10$$

There are looping terms like $\Omega \to \Omega \to \Omega \to \cdots$
Fix a program $\texttt{always\_1} \coloneqq \lambda x.1$. How to compute $\texttt{always\_1}(\Omega)$?

Call-by-Name

Call-by-Value



Feed any argument to any function

Only feed functions with values

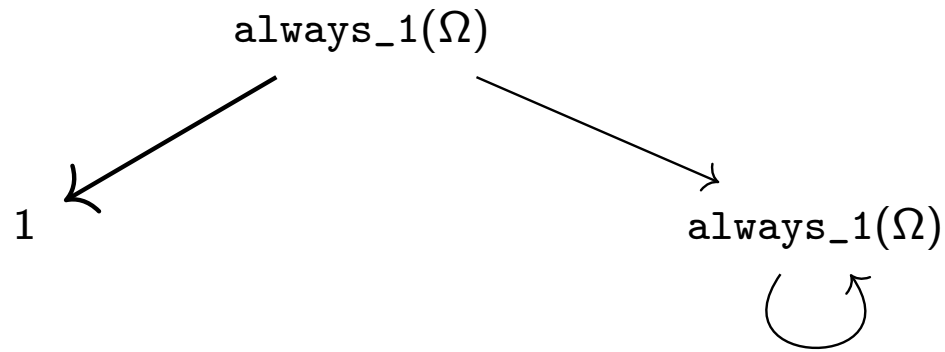
Contextual Equivalence depends on the Evaluation Paradigm

$$10 \xleftarrow{\text{CbN}} \texttt{double(2+3)} \xrightarrow{\text{CbV}} 10$$

There are looping terms like $\Omega \to \Omega \to \Omega \to \cdots$
Fix a program $\texttt{always\_1} \coloneqq \lambda x.1$. How to compute $\texttt{always\_1}(\Omega)$?

Call-by-Name

Call-by-Value



Feed any argument to any function

Only feed functions with values

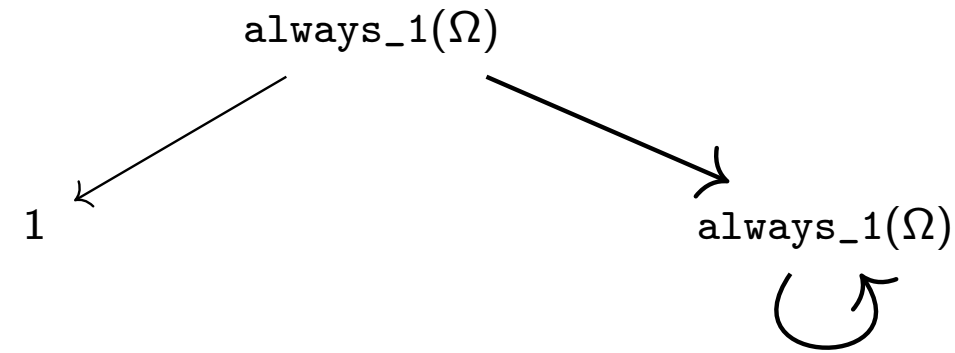
Contextual Equivalence depends on the Evaluation Paradigm

$$10 \xleftarrow{\text{CbN}} \texttt{double(2+3)} \xrightarrow{\text{CbV}} 10$$

There are looping terms like $\Omega \to \Omega \to \Omega \to \cdots$
Fix a program $\texttt{always\_1} \coloneqq \lambda x.1$. How to compute $\texttt{always\_1}(\Omega)$?

Call-by-Name

Call-by-Value



Feed any argument to any function

Only feed functions with values

Normal Form Bisimulations

How do we prove equivalence of (co)recursive programs?
Typically: equating fixpoint combinators

- **"Infinitely" Inspecting the Code:**

A program

```
let rec f :=
                  f(rec f).
```

Another program

```
let rec' f :=
              f(f(rec' f).
```

An (infinitary) semantics of these programs: $\lambda \mathtt{f}.\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{f}\ldots)))$

Building and Comparing infinite structures is done by **coinduction**.

$$\mathrm{BT}(\mathtt{rec}) = \lambda f.f \quad = \mathrm{BT}(\mathtt{rec'})$$
$$\Big|$$
$$f$$
$$\Big|$$
$$f$$
$$\vdots$$

Normal Form Bisimulations

How do we prove equivalence of (co)recursive programs?
Typically: equating fixpoint combinators

- **"Infinitely" Inspecting the Code:**

A program

```
let rec f :=
                  f(rec f).
```

Another program

```
let rec' f :=
              f(f(rec' f).
```

An (infinitary) semantics of these programs: $\lambda \mathtt{f}.\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{f}\ldots)))$

Building and Comparing infinite structures is done by **coinduction**.

$$\mathrm{BT}(\mathtt{rec}) = \lambda f.f \quad = \mathrm{BT}(\mathtt{rec'})$$
$$|$$
$$f$$
$$|$$
$$f$$
$$\vdots$$

Normal Form Bisimulations

How do we prove equivalence of (co)recursive programs?
Typically: equating fixpoint combinators

- **"Infinitely" Inspecting the Code:**

A program

```
let rec f :=
                  f(rec f).
```

Another program

```
let rec' f :=
              f(f(rec' f).
```

An (infinitary) semantics of these programs: $\lambda \mathtt{f}.\mathtt{f}(\mathtt{f}(\mathtt{f}\ldots))$

Building and Comparing infinite structures is done by **coinduction**.

$$\mathsf{BT}(\mathtt{rec}) = \quad \lambda f.f \quad = \mathsf{BT}(\mathtt{rec'})$$
$$\Big|$$
$$f$$
$$\Big|$$
$$f$$
$$\vdots$$

(In)Equational Theories

Fix an evaluation strategy $\mathtt{s}$ within a language of terms $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, $\rightarrow_{\mathtt{s}} \subseteq \rightarrow_{\mathcal{L}}$.

An inequational theory $=_{\mathcal{T}}$ for $\mathtt{s}$ is:

1. an *Equivalence Relation* on terms, $=_{\mathcal{T}} \subseteq \mathcal{L} \times \mathcal{L}$;
2. *Context-Closed*:
$$\text{if } t =_{\mathcal{T}} u \text{ then } \forall C,\ C\langle t \rangle =_{\mathcal{T}} C\langle u \rangle;$$
3. *Invariant under Computation*:
$$\text{if } t =_{\mathcal{L}} u \text{ then } t =_{\mathcal{T}} u.$$

(In)Equational Theories

Fix an evaluation strategy $\mathtt{s}$ within a language of terms $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, $\rightarrow_{\mathtt{s}} \subseteq \rightarrow_{\mathcal{L}}$.

An inequational theory $=_{\mathcal{T}}$ for $\mathtt{s}$ is:

1. an *Equivalence Relation* on terms, $=_{\mathcal{T}} \subseteq \mathcal{L} \times \mathcal{L}$;
2. *Context-Closed*:
$$\text{if } t =_{\mathcal{T}} u \text{ then } \forall C,\ C\langle t \rangle =_{\mathcal{T}} C\langle u \rangle;$$
3. *Invariant under Computation*:
$$\text{if } t =_{\mathcal{L}} u \text{ then } t =_{\mathcal{T}} u.$$

(In)Equational Theories

Fix an evaluation strategy $\mathtt{s}$ within a language of terms $(\mathcal{L}, \to_{\mathcal{L}})$, $\to_{\mathtt{s}} \subseteq \to_{\mathcal{L}}$.

An inequational theory $\leq_{\mathcal{T}}$ for $\mathtt{s}$ is:

1. a Preorder on terms, $\leq_{\mathcal{T}} \subseteq \mathcal{L} \times \mathcal{L}$;
2. *Context-Closed*:

$$\text{if } t \leq_{\mathcal{T}} u \text{ then } \forall C,\ C\langle t \rangle \leq_{\mathcal{T}} C\langle u \rangle;$$

3. *Invariant under Computation*:

$$\text{if } t =_{\mathcal{L}} u \text{ then } t \leq_{\mathcal{T}} u.$$

(In)Equational Theories

Fix an evaluation strategy $\mathtt{s}$ within a language of terms $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, $\rightarrow_{\mathtt{s}} \subseteq \rightarrow_{\mathcal{L}}$.

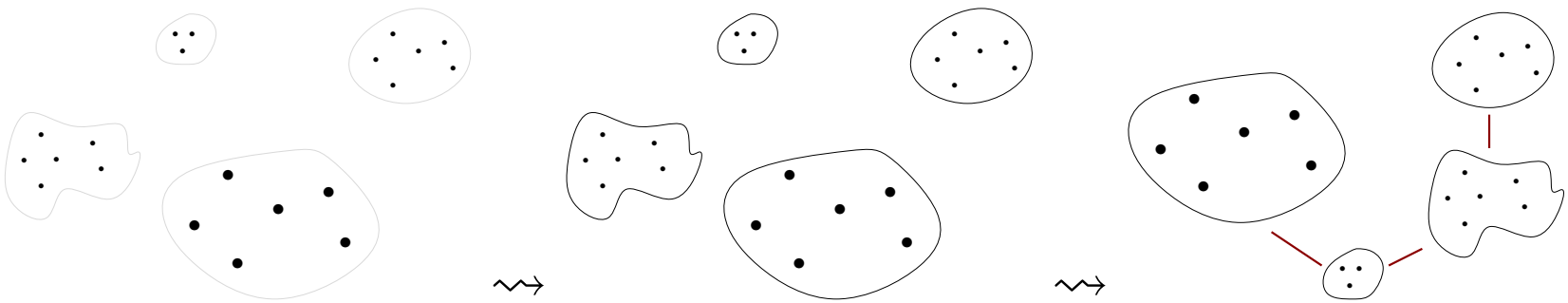
An inequational theory $\leq_{\mathcal{T}}$ for $\mathtt{s}$ is:

1. a Preorder on terms, $\leq_{\mathcal{T}} \subseteq \mathcal{L} \times \mathcal{L}$;
2. *Context-Closed*:

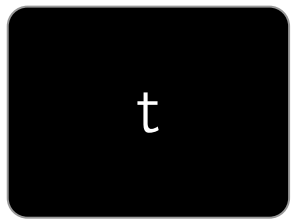
$$\text{if } t \leq_{\mathcal{T}} u \text{ then } \forall C,\ C\langle t \rangle \leq_{\mathcal{T}} C\langle u \rangle;$$

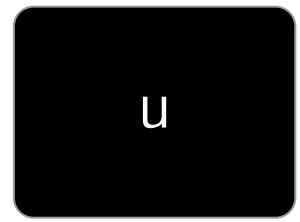
3. *Invariant under Computation*:

$$\text{if } t =_{\mathcal{L}} u \text{ then } t \leq_{\mathcal{T}} u.$$



Contextual Equivalence and Preorder





When are two λ-terms equivalent? [Mor68]

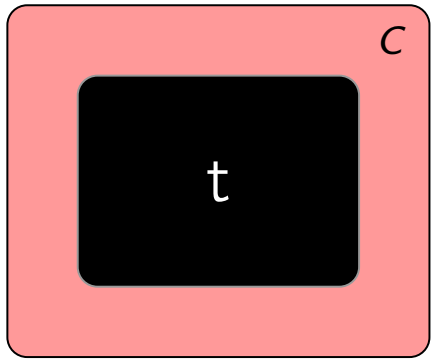
$$t \equiv_{\mathbf{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow \;\Leftrightarrow\; C\langle u\rangle \Downarrow\,]$$

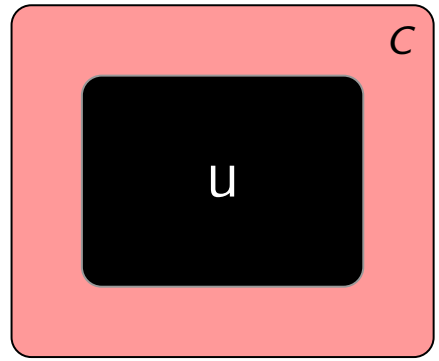
$$t \sqsubseteq_{\mathbf{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow \;\Rightarrow\; C\langle u\rangle \Downarrow\,]$$

The s-contextual preorder is generally an inequational theory.

Sufficient conditions on s and $\mathcal{L}$ are provided in the thesis.

Contextual Equivalence and Preorder





When are two λ-terms equivalent? [Mor68]

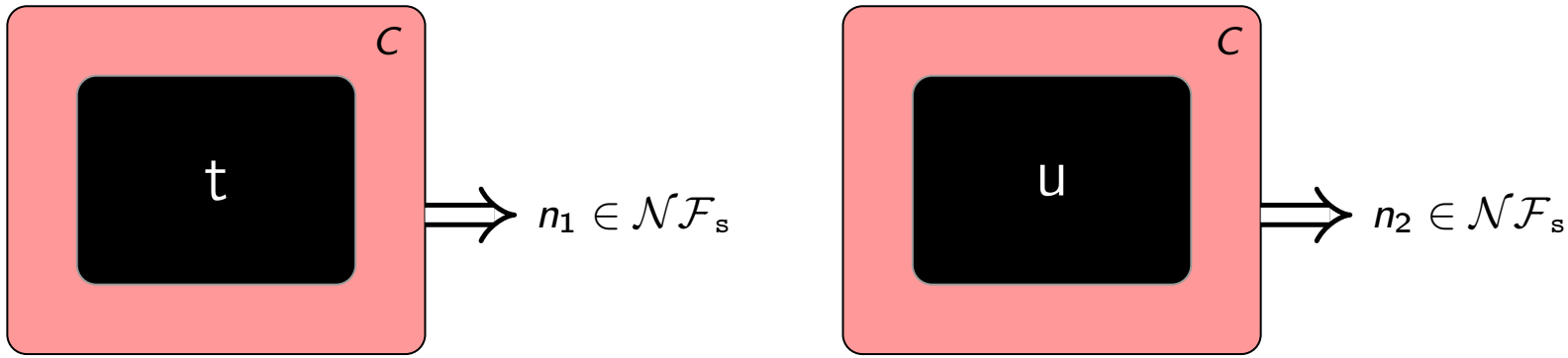
$$t \equiv_{\mathbf{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow \quad \Leftrightarrow \quad C\langle u\rangle \Downarrow\,]$$

$$t \sqsubseteq_{\mathbf{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow \quad \Rightarrow \quad C\langle u\rangle \Downarrow\,]$$

The s-contextual preorder is generally an inequational theory.

Sufficient conditions on s and $\mathcal{L}$ are provided in the thesis.

Contextual Equivalence and Preorder



When are two λ-terms equivalent? [Mor68]

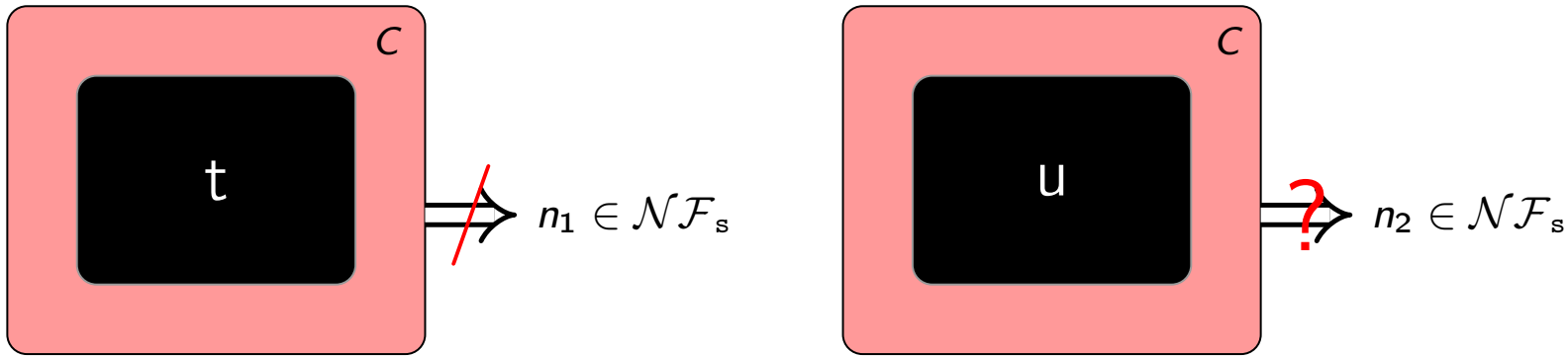
$$t \equiv_{\mathbf{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow_{\mathbf{s}} \quad \Leftrightarrow \quad C\langle u\rangle \Downarrow_{\mathbf{s}} \,]$$

$$t \sqsubseteq_{\mathbf{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow_{\mathbf{s}} \quad \Rightarrow \quad C\langle u\rangle \Downarrow_{\mathbf{s}} \,]$$

The s-contextual preorder is generally an inequational theory.

Sufficient conditions on s and $\mathcal{L}$ are provided in the thesis.

Contextual Equivalence and Preorder



When are two λ-terms equivalent? [Mor68]

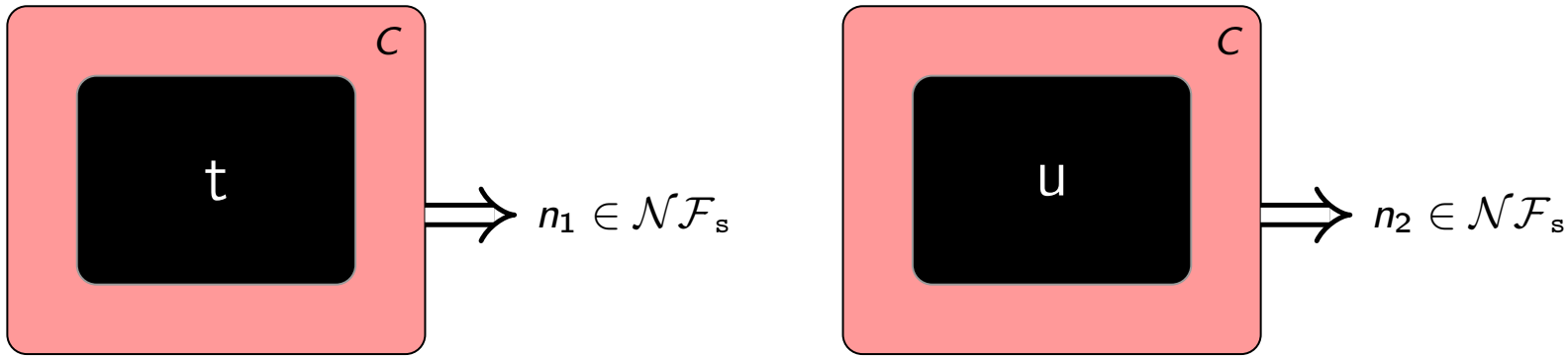
$$t \equiv_{\mathrm{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\; C\langle t\rangle \Downarrow_{\mathrm{s}} \quad \Leftrightarrow \quad C\langle u\rangle \Downarrow_{\mathrm{s}} \;]$$

$$t \sqsubseteq_{\mathrm{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\; C\langle t\rangle \Downarrow_{\mathrm{s}} \quad \Rightarrow \quad C\langle u\rangle \Downarrow_{\mathrm{s}} \;]$$

The s-contextual preorder is generally an inequational theory.

Sufficient conditions on s and $\mathcal{L}$ are provided in the thesis.

Contextual Equivalence and Preorder



When are two λ-terms equivalent? [Mor68]

$$t \equiv_{\mathrm{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow_{\mathrm{s}} \quad \Leftrightarrow \quad C\langle u\rangle \Downarrow_{\mathrm{s}} \,]$$

$$t \sqsubseteq_{\mathrm{s}}^{\mathrm{octx}} u \quad \text{if for all contexts } C\,. \quad [\, C\langle t\rangle \Downarrow_{\mathrm{s}} \quad \Rightarrow \quad C\langle u\rangle \Downarrow_{\mathrm{s}} \,]$$

The s-contextual preorder is generally an inequational theory.

Sufficient conditions on s and $\mathcal{L}$ are provided in the thesis.

Part I: Overview of Program Equivalences

Definable Type Equivalence
Operational Game Semantics
Weak Head Ineq. Theories
Interaction Improvement

Inequational Theories 🗋 FoSSaCs 2024

Light and Co Genericity

Maximality

Formalized CbN development 🗋 ITP 2025, formalized in Abella

Improvement theories

Normal form bisimulations
(and Böhm trees)
Applicative bisimulations

Type Equivalence (for CbN)

Part II:
A Pilgrimage into CbN

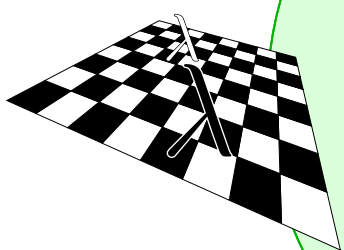
🗋 POPL 2025

Interaction Equivalence

Checkers Calculus

Ctx. Characterization of
Böhm tree equivalence

Checkers Relational Semantics



🗋 TYPES
PostProc 2025

Type Equivalence for lazy CbN

Part III:
A CbV Odyssey

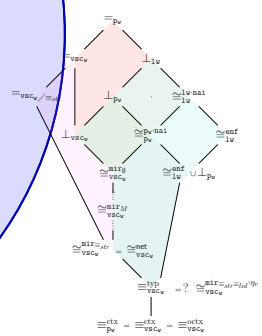
Theory of Call-by-Value
Light and Co CbV Genericity
Call-by-Silly

🗋 FoSSaCs 2024

🗋 FSCD 2024

Various CbV Normal form bisimulations

Type Equivalence for CbV



🗋 TechRep 2024


James Hiram Morris.
Lambda-calculus Models of Programming Languages.
PhD thesis, Massachusetts Institute of Technology, 1968.


Christopher P. Wadsworth.
Semantics and pragmatics of the lambda-calculus.
PhD Thesis, University of Oxford, 1971.