# Animating MRBNFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL

Jan van Brügge, Andrei Popescu, Dmitriy Traytel

## The problem

**Represent syntax in ITPs**

| | | | |
|---|---|---|---|
| $t$ | $::=$ | | Terms |
| | | $x$ | Variable |
| | $\mid$ | $\lambda x.\ t$ | Abstraction |
| | $\mid$ | $t\ t$ | Application |

## The problem

| | | | |
|---|---|---|---|
| $t$ | $::=$ | | Terms |
| | | $x$ | Variable |
| | $\mid$ | $\lambda x.\ t$ | Abstraction |
| | $\mid$ | $t\ t$ | Application |

**Represent syntax in ITPs**

- **alpha-equivalence**

## The problem

| | | | |
|---|---|---|---|
| $t$ | $::=$ | | Terms |
| | | $x$ | Variable |
| | $\mid$ | $\lambda x.\ t$ | Abstraction |
| | $\mid$ | $t\ t$ | Application |

**Represent syntax in ITPs**

- **alpha-equivalence**
- **free variables**

## The problem

| | | | |
|---|---|---|---|
| $t$ | $::=$ | | Terms |
| | | $x$ | Variable |
| | $\mid$ | $\lambda x.\ t$ | Abstraction |
| | $\mid$ | $t\ t$ | Application |

**Represent syntax in ITPs**

- **alpha-equivalence**
- **free variables**
- **induction**

## The problem

| | | |
|---|---|---|
| $t$ | $::=$ | Terms |
| | $x$ | Variable |
| $\mid$ | $\lambda x.\ t$ | Abstraction |
| $\mid$ | $t\ t$ | Application |

**Represent syntax in ITPs**

- **alpha-equivalence**
- **free variables**
- **induction**
- **renaming**

## The problem

$$
\begin{array}{llll}
t & ::= & & \text{Terms} \\
 & & x & \text{Variable} \\
 & | & \lambda x.\ t & \text{Abstraction} \\
 & | & t\ t & \text{Application}
\end{array}
$$

**Represent syntax in ITPs**

- **alpha-equivalence**
- **free variables**
- **induction**
- **renaming**
- **substitution**

## The problem

| | | | |
|---|---|---|---|
| $t$ | $::=$ | | Terms |
| | | $x$ | Variable |
| | $\mid$ | $\lambda x.\ t$ | Abstraction |
| | $\mid$ | $t\ t$ | Application |

**Represent syntax in ITPs**

- **alpha-equivalence**
- **free variables**
- **induction**
- **renaming**
- **substitution**
- **arbitrary functions**

## Demo

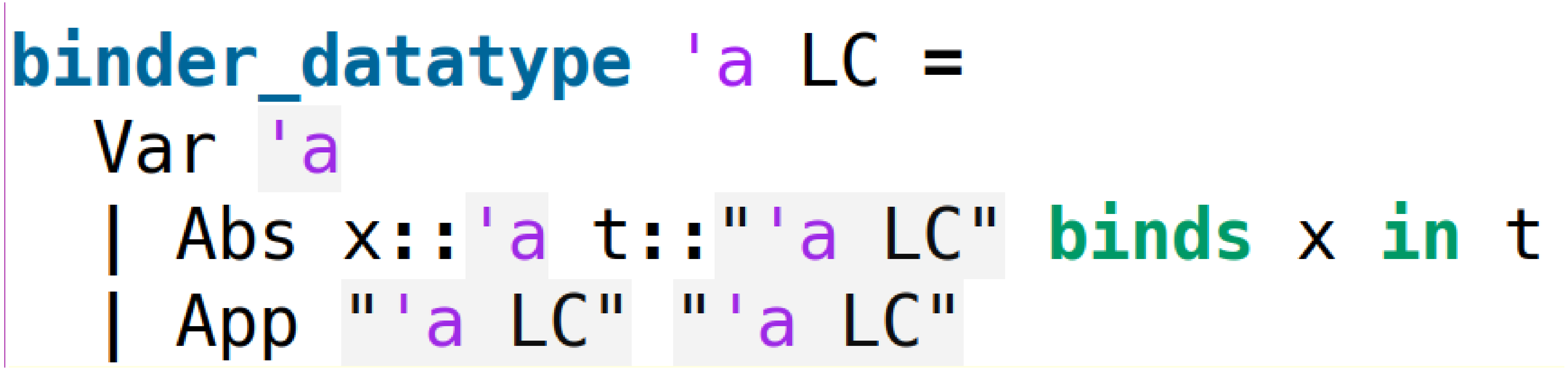
Switch to Isabelle

## Demo

Switch to Isabelle

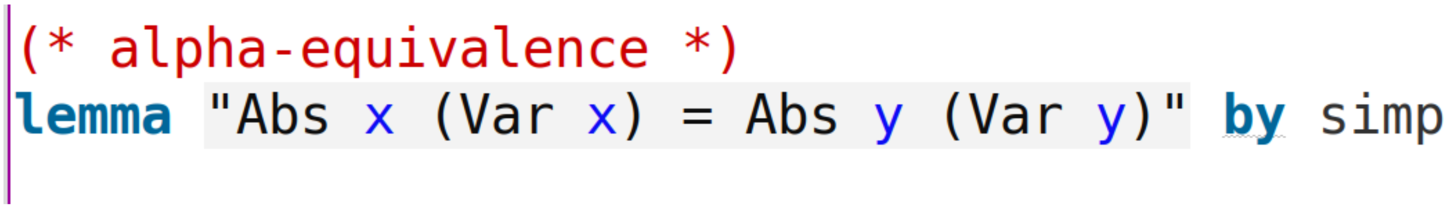
Goto next

## Demo (Backup Slides)



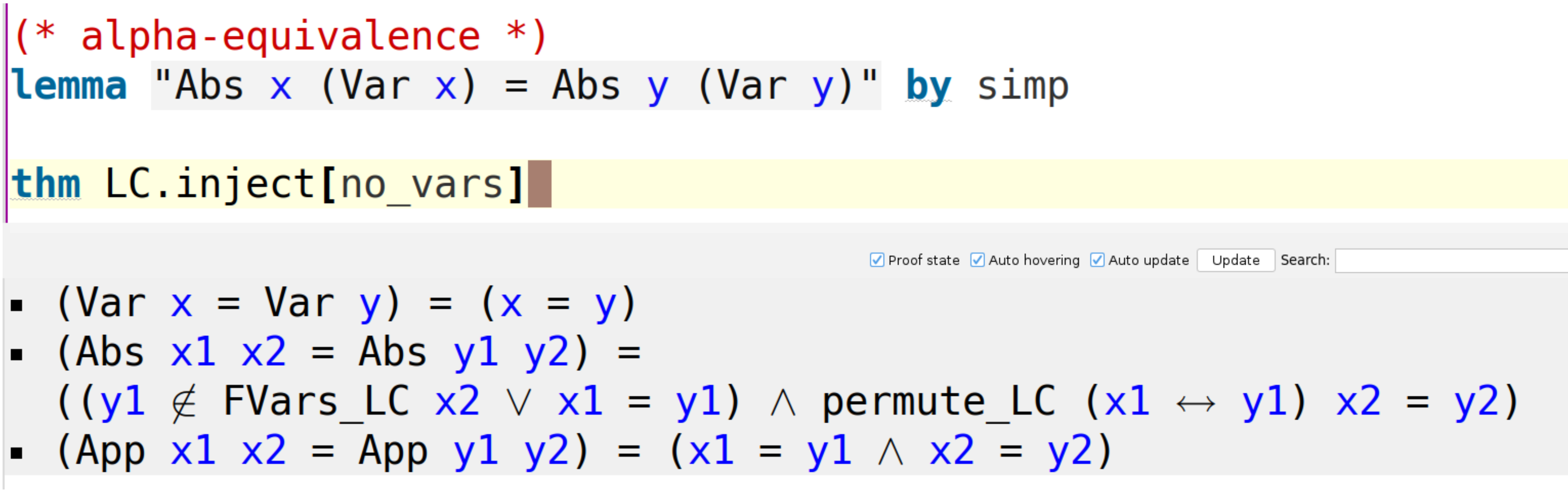
```
binder_datatype 'a LC =
  Var 'a
  | Abs x::'a t::"'a LC" binds x in t
  | App "'a LC" "'a LC"
```

## Demo (Backup Slides)



```
(* alpha-equivalence *)
lemma "Abs x (Var x) = Abs y (Var y)" by simp
```

## Demo (Backup Slides)



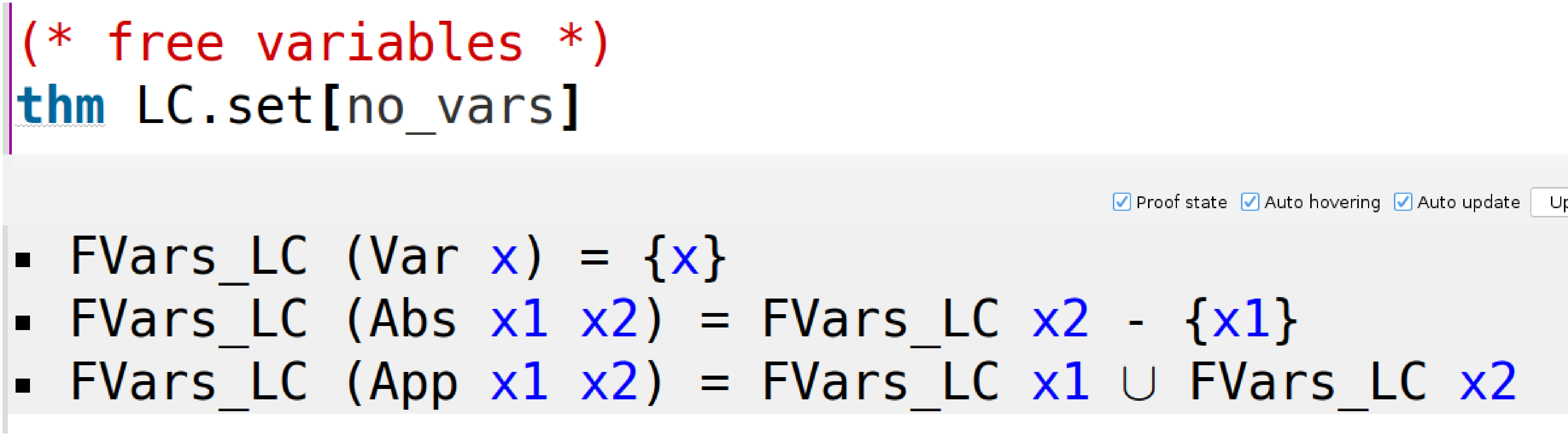
```
(* alpha-equivalence *)
lemma "Abs x (Var x) = Abs y (Var y)" by simp

thm LC.inject[no_vars]
```

☑ Proof state ☑ Auto hovering ☑ Auto update  Update  Search:

```
▪ (Var x = Var y) = (x = y)
▪ (Abs x1 x2 = Abs y1 y2) =
  ((y1 ∉ FVars_LC x2 ∨ x1 = y1) ∧ permute_LC (x1 ↔ y1) x2 = y2)
▪ (App x1 x2 = App y1 y2) = (x1 = y1 ∧ x2 = y2)
```

## Demo (Backup Slides)

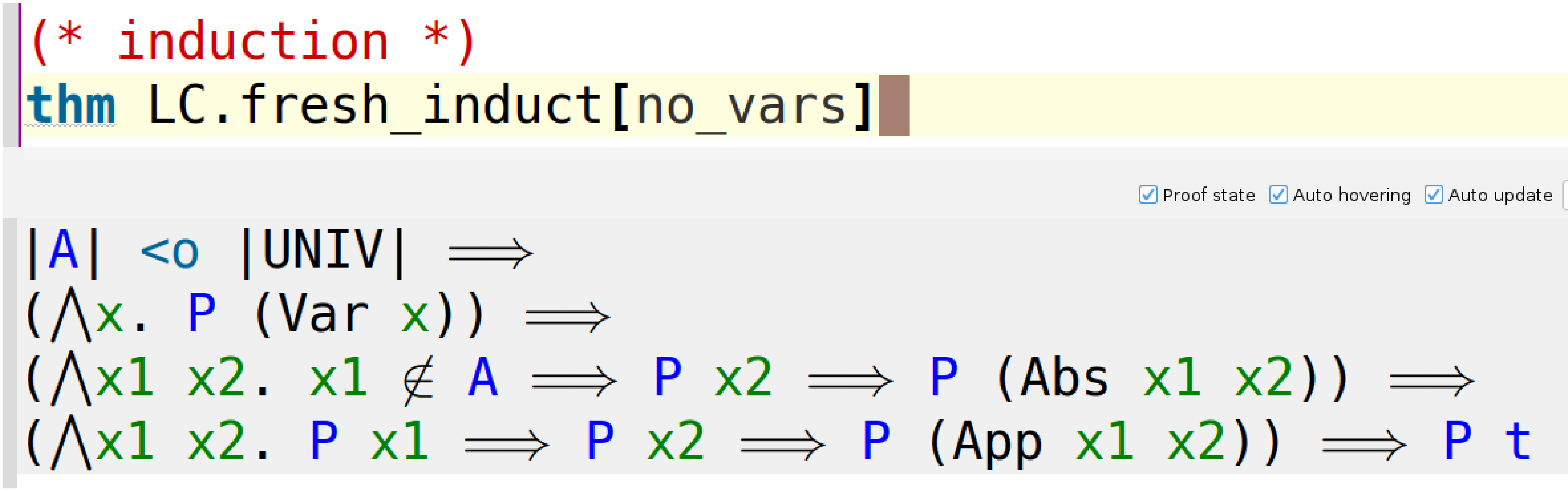


```
(* free variables *)
thm LC.set[no_vars]
```

✓ Proof state ✓ Auto hovering ✓ Auto update

```
▪ FVars_LC (Var x) = {x}
▪ FVars_LC (Abs x1 x2) = FVars_LC x2 - {x1}
▪ FVars_LC (App x1 x2) = FVars_LC x1 ∪ FVars_LC x2
```

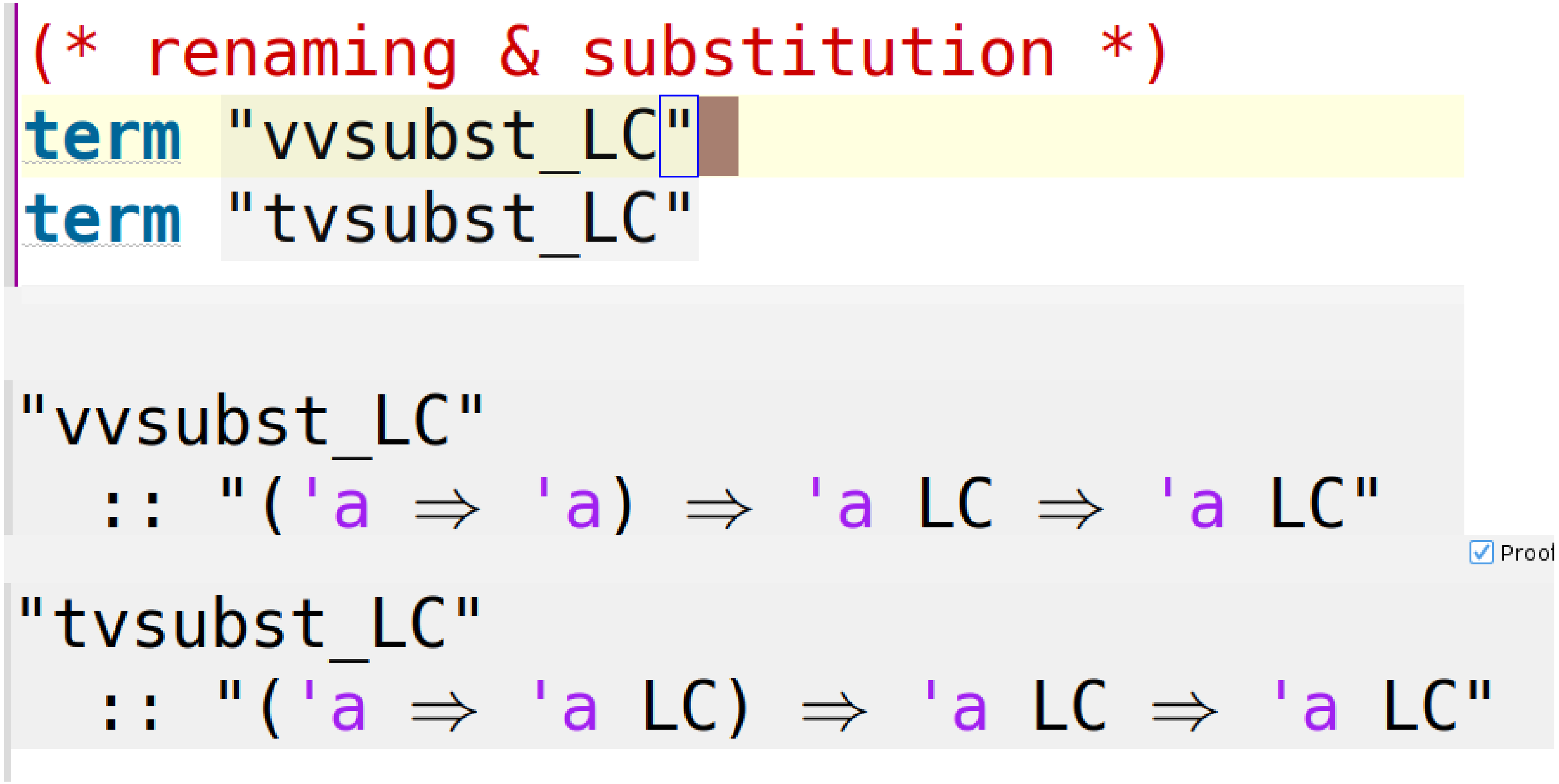
# Demo (Backup Slides)



```
(* induction *)
thm LC.fresh_induct[no_vars]
```

```
|A| <o |UNIV| ⟹
(⋀x. P (Var x)) ⟹
(⋀x1 x2. x1 ∉ A ⟹ P x2 ⟹ P (Abs x1 x2)) ⟹
(⋀x1 x2. P x1 ⟹ P x2 ⟹ P (App x1 x2)) ⟹ P t
```

## Demo (Backup Slides)

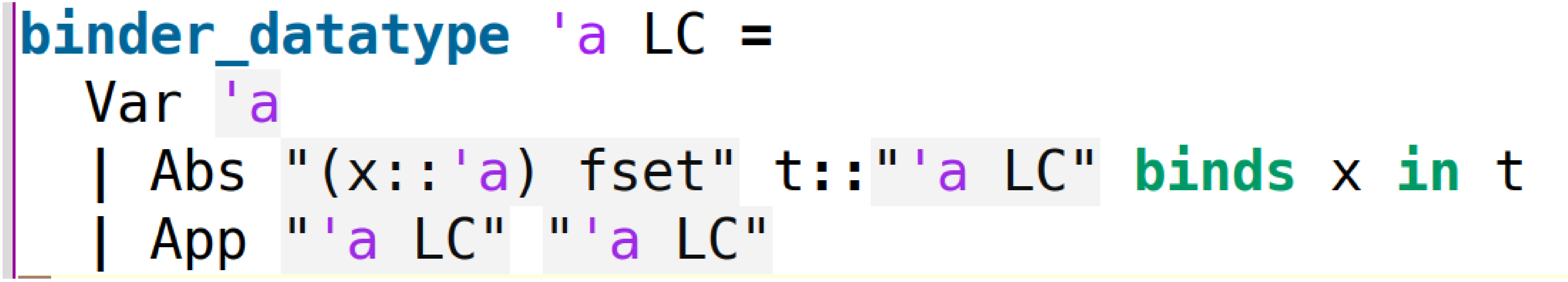


```
(* renaming & substitution *)
term "vvsubst_LC"
term "tvsubst_LC"
```

```
"vvsubst_LC"
  :: "('a ⇒ 'a) ⇒ 'a LC ⇒ 'a LC"
```

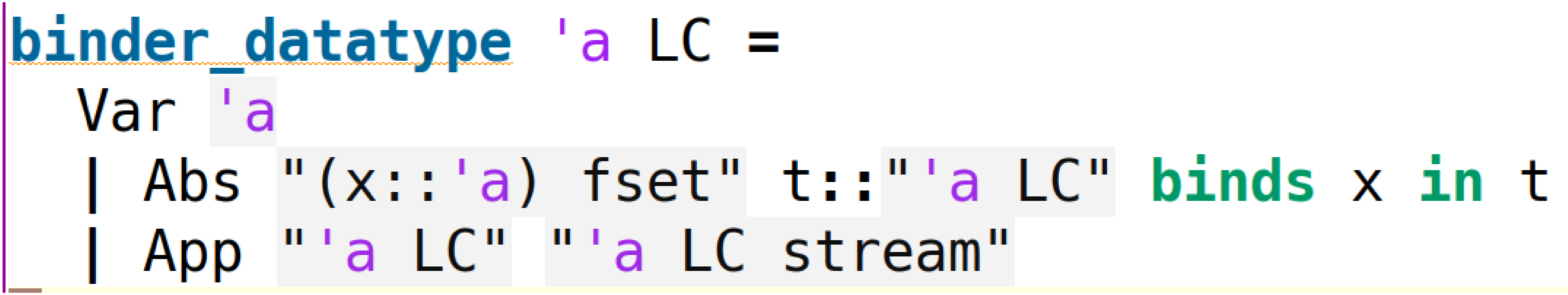
```
"tvsubst_LC"
  :: "('a ⇒ 'a LC) ⇒ 'a LC ⇒ 'a LC"
```

## Demo (Backup Slides)



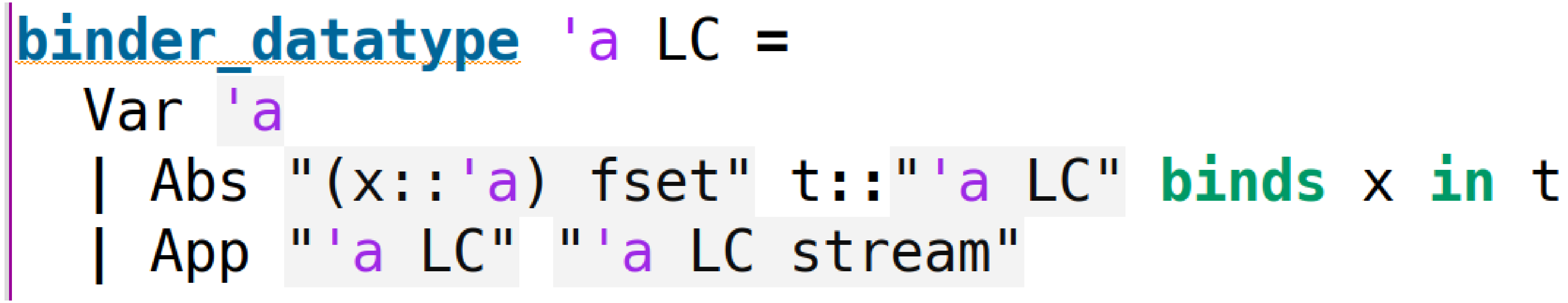
```
binder_datatype 'a LC =
  Var 'a
  | Abs "(x::'a) fset" t::"'a LC" binds x in t
  | App "'a LC" "'a LC"
```

## Demo (Backup Slides)



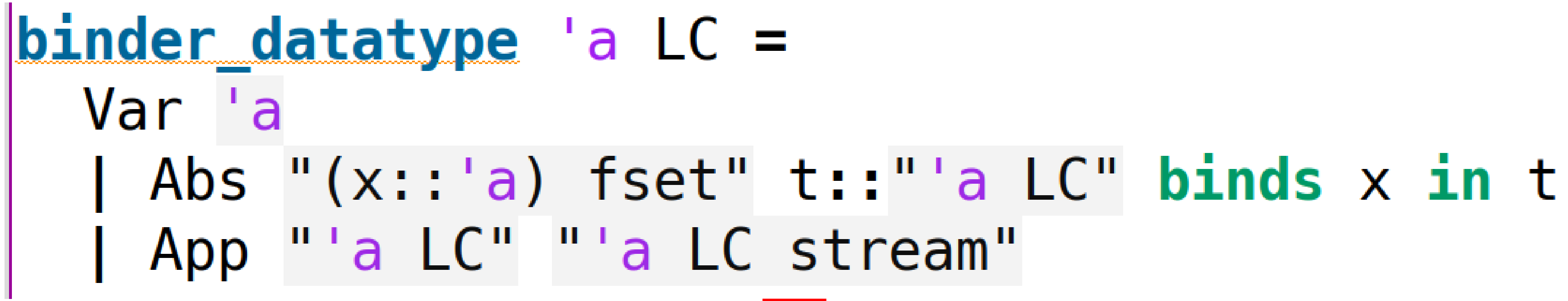
```
binder_datatype 'a LC =
  Var 'a
  | Abs "(x::'a) fset" t::"'a LC" binds x in t
  | App "'a LC" "'a LC stream"
```

## How does it work?

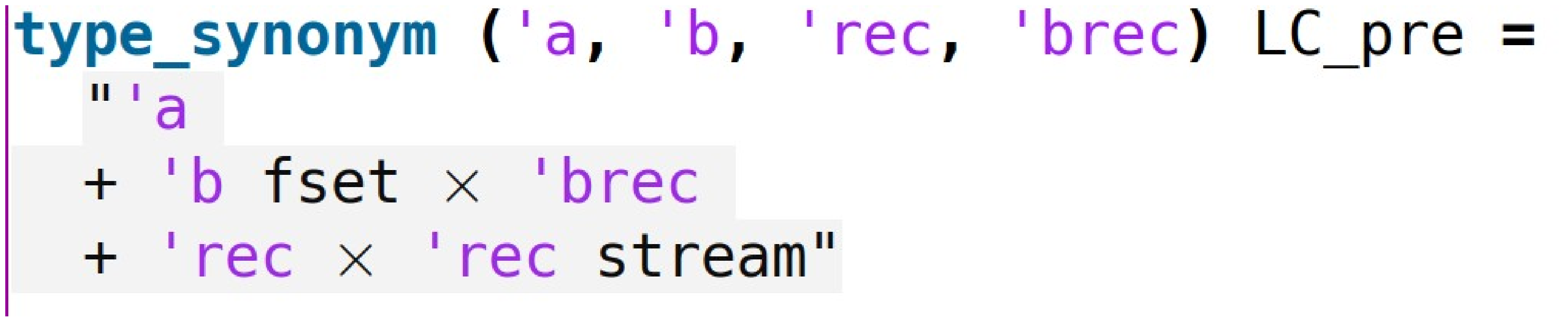


```
binder_datatype 'a LC =
  Var 'a
  | Abs "(x::'a) fset" t::"'a LC" binds x in t
  | App "'a LC" "'a LC stream"
```

## How does it work?

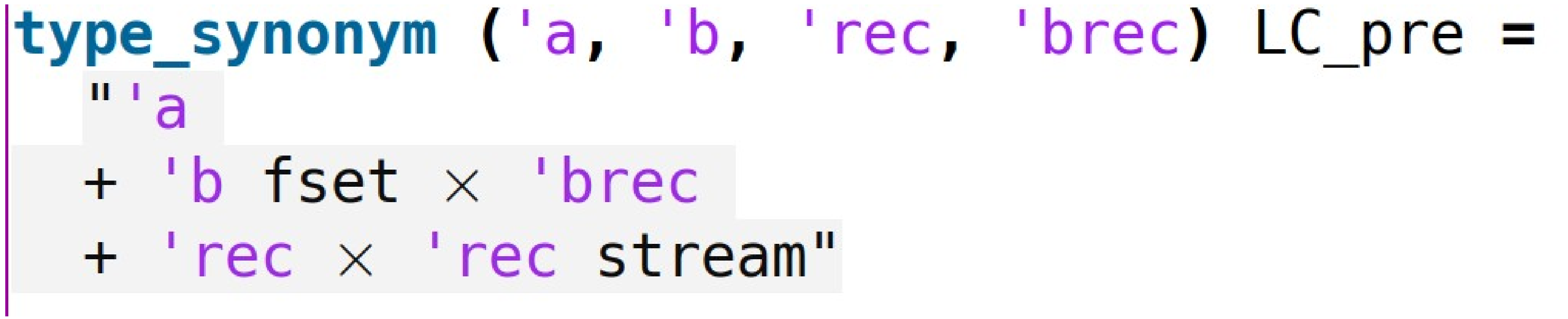


```
binder_datatype 'a LC =
    Var 'a
  | Abs "(x::'a) fset" t::"'a LC" binds x in t
  | App "'a LC" "'a LC stream"
```



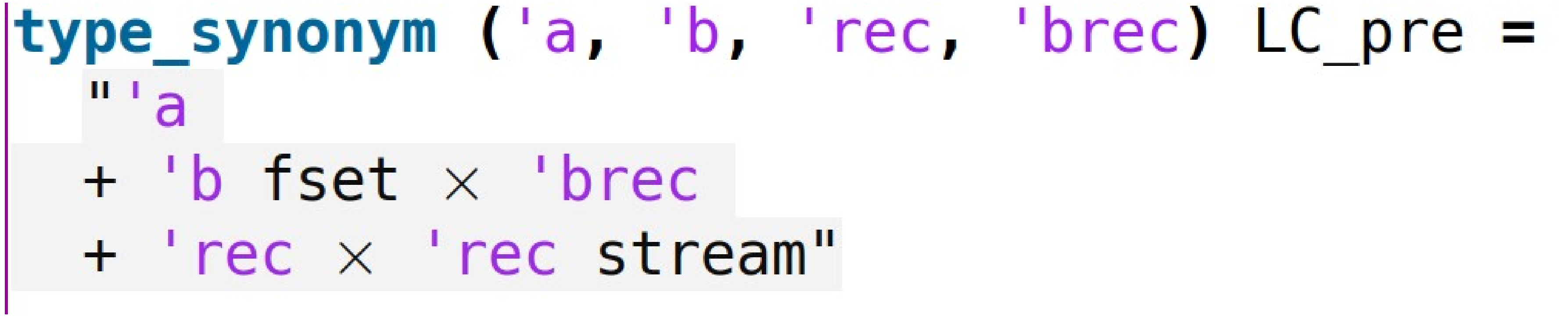
```
type_synonym ('a, 'b, 'rec, 'brec) LC_pre =
  "'a
  + 'b fset × 'brec
  + 'rec × 'rec stream"
```

## How does it work?

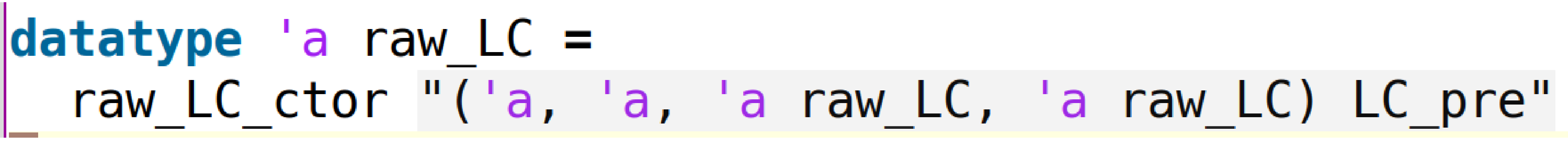


```
type_synonym ('a, 'b, 'rec, 'brec) LC_pre =
  "'a
  + 'b fset × 'brec
  + 'rec × 'rec stream"
```

## How does it work?

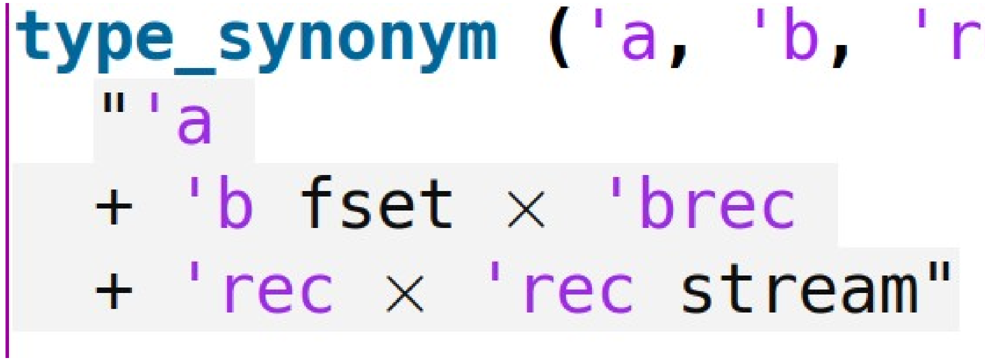


```
type_synonym ('a, 'b, 'rec, 'brec) LC_pre =
  "'a
  + 'b fset × 'brec
  + 'rec × 'rec stream"
```



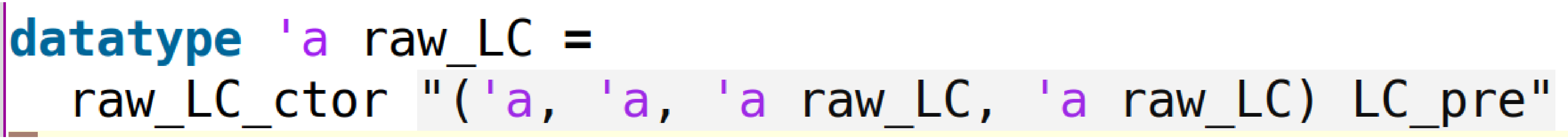
```
datatype 'a raw_LC =
  raw_LC_ctor "('a, 'a, 'a raw_LC, 'a raw_LC) LC_pre"
```

# How does it work?



```
type_synonym ('a, 'b, 'rec, 'brec) LC_pre =
  "'a
  + 'b fset × 'brec
  + 'rec × 'rec stream"
```

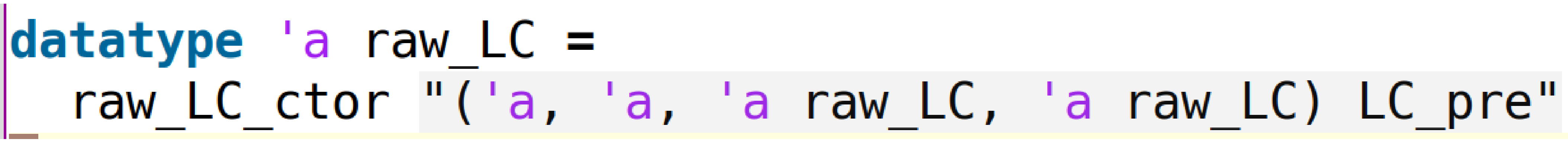
⬇ *



```
datatype 'a raw_LC =
  raw_LC_ctor "('a, 'a, 'a raw_LC, 'a raw_LC) LC_pre"
```

* requires automated proof that LC_pre is a MRBNF (see paper for details)

## How does it work?



```
datatype 'a raw_LC =
  raw_LC_ctor "('a, 'a, 'a raw_LC, 'a raw_LC) LC_pre"
```

## How does it work?

```
datatype 'a raw_LC =
  raw_LC_ctor "('a, 'a, 'a raw_LC, 'a raw_LC) LC_pre"
```

```
typedef 'a LC = "(UNIV :: 'a raw_LC set)
    // { (x, y). alpha_LC x y }"
```

## How does it work?

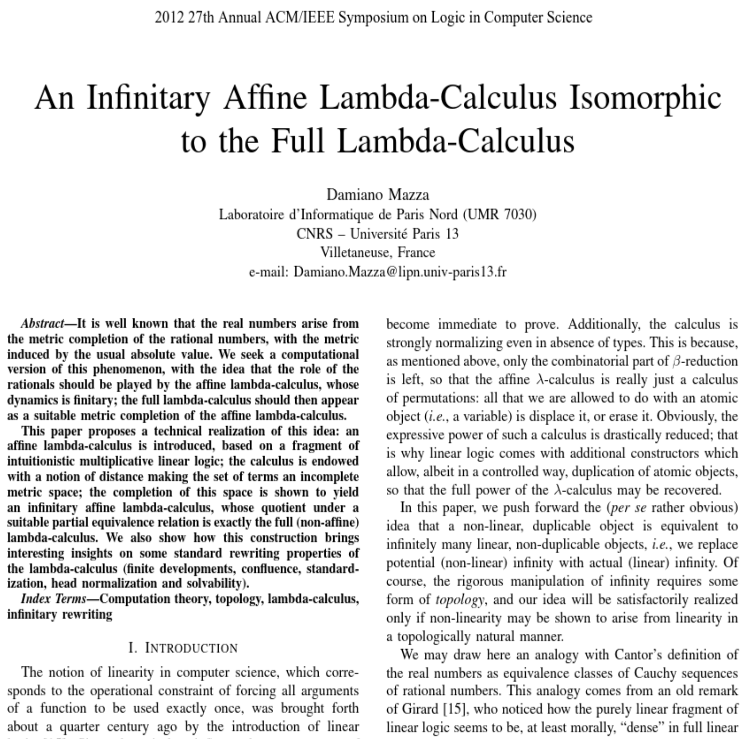
```
locale REC =
fixes Pmap :: "('a ⇒ 'a) ⇒ 'p ⇒ 'p"
  and PFVars :: "'p ⇒ 'a set"
  and validP :: "'p ⇒ bool"

  and avoiding_set :: "'a set"

  and Umap :: "('a ⇒ 'a) ⇒ 'u ⇒ 'u"
  and UFVars :: "'u ⇒ 'a set"
  and validU :: "'u ⇒ bool"
  and Uctor ::
   "('a, 'a, 'a LC × ('p ⇒ 'u), 'a LC × ('p ⇒ 'u)) LC_pre ⇒ 'p ⇒ 'u"
```

## More in the paper!

Case study 1:
Mazza’s Isomorphism



2012 27th Annual ACM/IEEE Symposium on Logic in Computer Science

### An Infinitary Affine Lambda-Calculus Isomorphic to the Full Lambda-Calculus

Damiano Mazza
Laboratoire d’Informatique de Paris Nord (UMR 7030)
CNRS – Université Paris 13
Villetaneuse, France
e-mail: Damiano.Mazza@lipn.univ-paris13.fr

***Abstract*—It is well known that the real numbers arise from the metric completion of the rational numbers, with the metric induced by the usual absolute value. We seek a computational version of this phenomenon, with the idea that the role of the rationals should be played by the affine lambda-calculus, whose dynamics is finitary; the full lambda-calculus should then appear as a suitable metric completion of the affine lambda-calculus.**

**This paper proposes a technical realization of this idea: an affine lambda-calculus is introduced, based on a fragment of intuitionistic multiplicative linear logic; the calculus is endowed with a notion of distance making the set of terms an incomplete metric space; the completion of this space is shown to yield an infinitary affine lambda-calculus, whose quotient under a suitable partial equivalence relation is exactly the full (non-affine) lambda-calculus. We also show how this construction brings interesting insights on some standard rewriting properties of the lambda-calculus (finite developments, confluence, standardization, head normalization and solvability).**

***Index Terms*—Computation theory, topology, lambda-calculus, infinitary rewriting**

#### I. Introduction

The notion of linearity in computer science, which corresponds to the operational constraint of forcing all arguments of a function to be used exactly once, was brought forth about a quarter century ago by the introduction of linear

become immediate to prove. Additionally, the calculus is strongly normalizing even in absence of types. This is because, as mentioned above, only the combinatorial part of $\beta$-reduction is left, so that the affine $\lambda$-calculus is really just a calculus of permutations: all that we are allowed to do with an atomic object (*i.e.*, a variable) is displace it, or erase it. Obviously, the expressive power of such a calculus is drastically reduced; that is why linear logic comes with additional constructors which allow, albeit in a controlled way, duplication of atomic objects, so that the full power of the $\lambda$-calculus may be recovered.

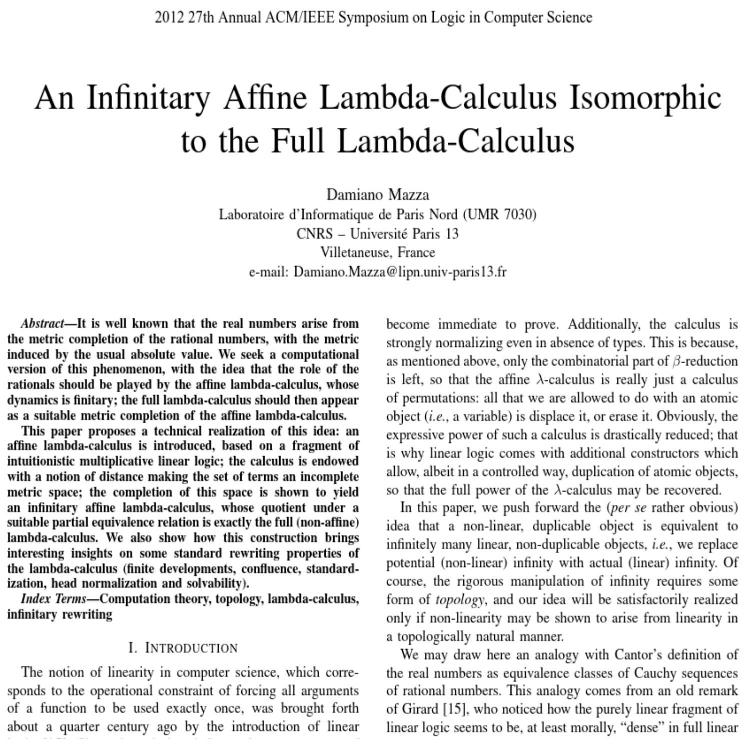
In this paper, we push forward the (*per se* rather obvious) idea that a non-linear, duplicable object is equivalent to infinitely many linear, non-duplicable objects, *i.e.*, we replace potential (non-linear) infinity with actual (linear) infinity. Of course, the rigorous manipulation of infinity requires some form of *topology*, and our idea will be satisfactorily realized only if non-linearity may be shown to arise from linearity in a topologically natural manner.

We may draw here an analogy with Cantor’s definition of the real numbers as equivalence classes of Cauchy sequences of rational numbers. This analogy comes from an old remark of Girard [15], who noticed how the purely linear fragment of linear logic seems to be, at least morally, “dense” in full linear

# More in the paper!

Case study 1:
Mazza’s Isomorphism

- Lambda binds (distinct) stream of variables



2012 27th Annual ACM/IEEE Symposium on Logic in Computer Science

An Infinitary Affine Lambda-Calculus Isomorphic to the Full Lambda-Calculus

Damiano Mazza
Laboratoire d’Informatique de Paris Nord (UMR 7030)
CNRS – Université Paris 13
Villetaneuse, France
e-mail: Damiano.Mazza@lipn.univ-paris13.fr

***Abstract*—It is well known that the real numbers arise from the metric completion of the rational numbers, with the metric induced by the usual absolute value. We seek a computational version of this phenomenon, with the idea that the role of the rationals should be played by the affine lambda-calculus, whose dynamics is finitary; the full lambda-calculus should then appear as a suitable metric completion of the affine lambda-calculus.**

**This paper proposes a technical realization of this idea: an affine lambda-calculus is introduced, based on a fragment of intuitionistic multiplicative linear logic; the calculus is endowed with a notion of distance making the set of terms an incomplete metric space; the completion of this space is shown to yield an infinitary affine lambda-calculus, whose quotient under a suitable partial equivalence relation is exactly the full (non-affine) lambda-calculus. We also show how this construction brings interesting insights on some standard rewriting properties of the lambda-calculus (finite developments, confluence, standardization, head normalization and solvability).**

***Index Terms*—Computation theory, topology, lambda-calculus, infinitary rewriting**

I. Introduction

The notion of linearity in computer science, which corresponds to the operational constraint of forcing all arguments of a function to be used exactly once, was brought forth about a quarter century ago by the introduction of linear

become immediate to prove. Additionally, the calculus is strongly normalizing even in absence of types. This is because, as mentioned above, only the combinatorial part of $\beta$-reduction is left, so that the affine $\lambda$-calculus is really just a calculus of permutations: all that we are allowed to do with an atomic object (*i.e.*, a variable) is displace it, or erase it. Obviously, the expressive power of such a calculus is drastically reduced; that is why linear logic comes with additional constructors which allow, albeit in a controlled way, duplication of atomic objects, so that the full power of the $\lambda$-calculus may be recovered.

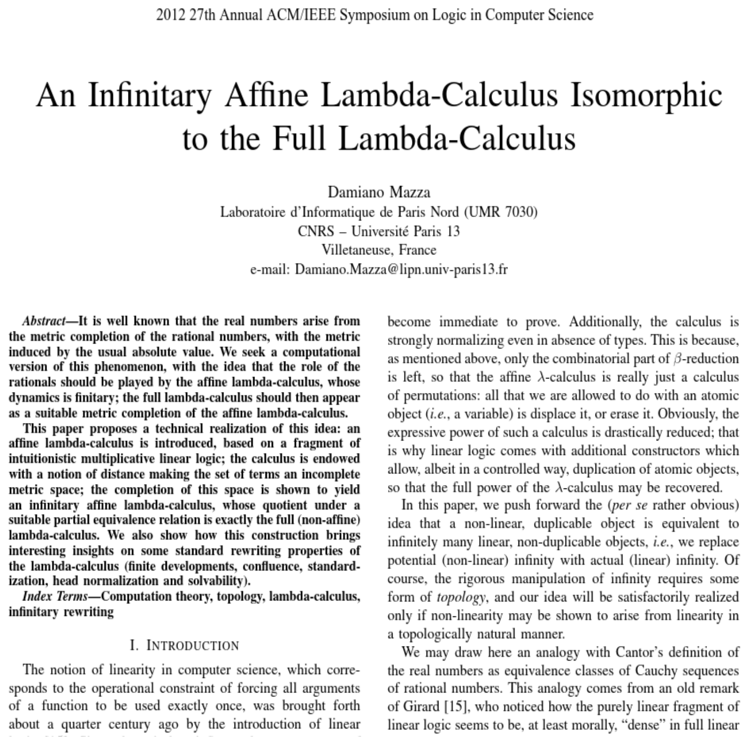
In this paper, we push forward the (*per se* rather obvious) idea that a non-linear, duplicable object is equivalent to infinitely many linear, non-duplicable objects, *i.e.*, we replace potential (non-linear) infinity with actual (linear) infinity. Of course, the rigorous manipulation of infinity requires some form of *topology*, and our idea will be satisfactorily realized only if non-linearity may be shown to arise from linearity in a topologically natural manner.

We may draw here an analogy with Cantor’s definition of the real numbers as equivalence classes of Cauchy sequences of rational numbers. This analogy comes from an old remark of Girard [15], who noticed how the purely linear fragment of linear logic seems to be, at least morally, “dense” in full linear

# More in the paper!

Case study 1:
Mazza’s Isomorphism

- Lambda binds (distinct) stream of variables
- Application applies stream of terms



2012 27th Annual ACM/IEEE Symposium on Logic in Computer Science

An Infinitary Affine Lambda-Calculus Isomorphic to the Full Lambda-Calculus

Damiano Mazza
Laboratoire d’Informatique de Paris Nord (UMR 7030)
CNRS – Université Paris 13
Villetaneuse, France
e-mail: Damiano.Mazza@lipn.univ-paris13.fr

***Abstract*—It is well known that the real numbers arise from the metric completion of the rational numbers, with the metric induced by the usual absolute value. We seek a computational version of this phenomenon, with the idea that the role of the rationals should be played by the affine lambda-calculus, whose dynamics is finitary; the full lambda-calculus should then appear as a suitable metric completion of the affine lambda-calculus.**

**This paper proposes a technical realization of this idea: an affine lambda-calculus is introduced, based on a fragment of intuitionistic multiplicative linear logic; the calculus is endowed with a notion of distance making the set of terms an incomplete metric space; the completion of this space is shown to yield an infinitary affine lambda-calculus, whose quotient under a suitable partial equivalence relation is exactly the full (non-affine) lambda-calculus. We also show how this construction brings interesting insights on some standard rewriting properties of the lambda-calculus (finite developments, confluence, standardization, head normalization and solvability).**

***Index Terms*—Computation theory, topology, lambda-calculus, infinitary rewriting**

I. Introduction

The notion of linearity in computer science, which corresponds to the operational constraint of forcing all arguments of a function to be used exactly once, was brought forth about a quarter century ago by the introduction of linear

become immediate to prove. Additionally, the calculus is strongly normalizing even in absence of types. This is because, as mentioned above, only the combinatorial part of $\beta$-reduction is left, so that the affine $\lambda$-calculus is really just a calculus of permutations: all that we are allowed to do with an atomic object (*i.e.*, a variable) is displace it, or erase it. Obviously, the expressive power of such a calculus is drastically reduced; that is why linear logic comes with additional constructors which allow, albeit in a controlled way, duplication of atomic objects, so that the full power of the $\lambda$-calculus may be recovered.

In this paper, we push forward the (*per se* rather obvious) idea that a non-linear, duplicable object is equivalent to infinitely many linear, non-duplicable objects, *i.e.*, we replace potential (non-linear) infinity with actual (linear) infinity. Of course, the rigorous manipulation of infinity requires some form of *topology*, and our idea will be satisfactorily realized only if non-linearity may be shown to arise from linearity in a topologically natural manner.

We may draw here an analogy with Cantor’s definition of the real numbers as equivalence classes of Cauchy sequences of rational numbers. This analogy comes from an old remark of Girard [15], who noticed how the purely linear fragment of linear logic seems to be, at least morally, “dense” in full linear

## More in the paper!

Case study 2:
POPLmark 2B

Mechanized Metatheory for the Masses:
The POPLMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

**Abstract.** How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

1 Introduction

## More in the paper!

Case study 2:
POPLmark 2B

- Case expressions use nested record patterns with distinct binders

**Mechanized Metatheory for the Masses:**
**The POPLMARK Challenge**

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

**Abstract.** How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

**1 Introduction**

## More in the paper!

Case study 2:
POPLmark 2B

- Case expressions use nested record patterns with distinct binders
- First solution that uses names

**Mechanized Metatheory for the Masses:**
**The POPLMark Challenge**

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

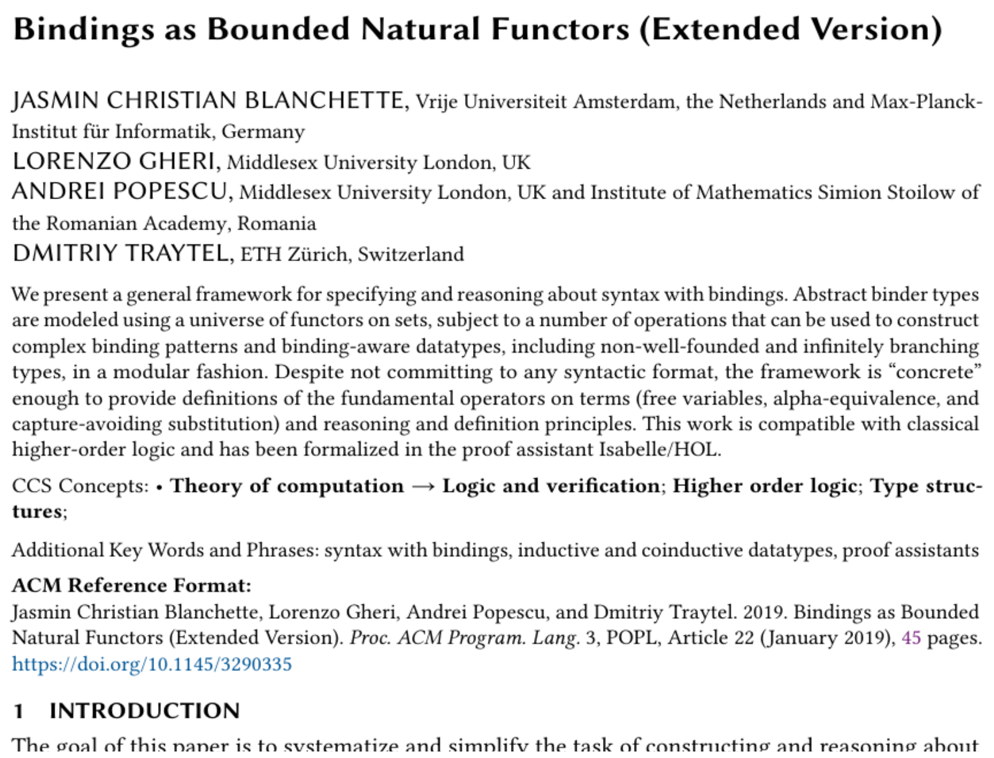
**Abstract.** How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

**1 Introduction**

## Short note on history

## Short note on history

- POPL19 paper by Blanchette et. al. introducing MRBNFs



**Bindings as Bounded Natural Functors (Extended Version)**

JASMIN CHRISTIAN BLANCHETTE, Vrije Universiteit Amsterdam, the Netherlands and Max-Planck-Institut für Informatik, Germany
LORENZO GHERI, Middlesex University London, UK
ANDREI POPESCU, Middlesex University London, UK and Institute of Mathematics Simion Stoilow of the Romanian Academy, Romania
DMITRIY TRAYTEL, ETH Zürich, Switzerland

We present a general framework for specifying and reasoning about syntax with bindings. Abstract binder types are modeled using a universe of functors on sets, subject to a number of operations that can be used to construct complex binding patterns and binding-aware datatypes, including non-well-founded and infinitely branching types, in a modular fashion. Despite not committing to any syntactic format, the framework is "concrete" enough to provide definitions of the fundamental operators on terms (free variables, alpha-equivalence, and capture-avoiding substitution) and reasoning and definition principles. This work is compatible with classical higher-order logic and has been formalized in the proof assistant Isabelle/HOL.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Higher order logic**; **Type structures**;

Additional Key Words and Phrases: syntax with bindings, inductive and coinductive datatypes, proof assistants

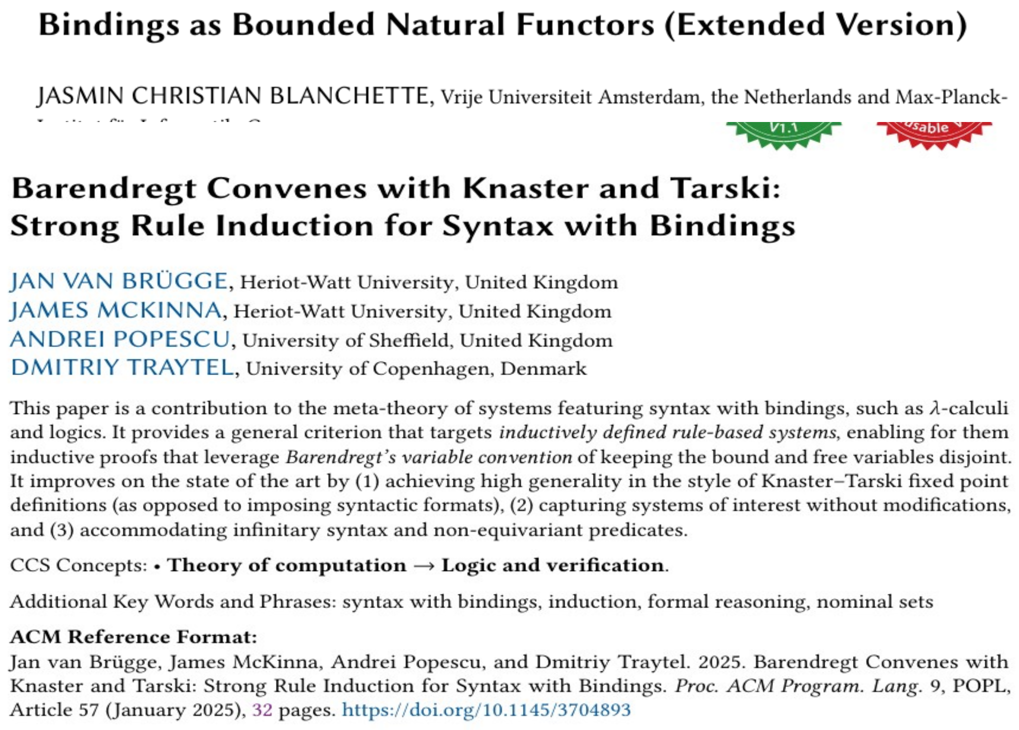
**ACM Reference Format:**
Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019. Bindings as Bounded Natural Functors (Extended Version). *Proc. ACM Program. Lang.* 3, POPL, Article 22 (January 2019), 45 pages. https://doi.org/10.1145/3290335

1 INTRODUCTION

The goal of this paper is to systematize and simplify the task of constructing and reasoning about

# Short note on history

- POPL19 paper by Blanchette et. al. introducing MRBNFs
- POPL25 paper on strong rule induction for inductive predicates



**Bindings as Bounded Natural Functors (Extended Version)**

JASMIN CHRISTIAN BLANCHETTE, Vrije Universiteit Amsterdam, the Netherlands and Max-Planck-

**Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings**

JAN VAN BRÜGGE, Heriot-Watt University, United Kingdom
JAMES MCKINNA, Heriot-Watt University, United Kingdom
ANDREI POPESCU, University of Sheffield, United Kingdom
DMITRIY TRAYTEL, University of Copenhagen, Denmark

This paper is a contribution to the meta-theory of systems featuring syntax with bindings, such as $\lambda$-calculi and logics. It provides a general criterion that targets *inductively defined rule-based systems*, enabling for them inductive proofs that leverage *Barendregt's variable convention* of keeping the bound and free variables disjoint. It improves on the state of the art by (1) achieving high generality in the style of Knaster–Tarski fixed point definitions (as opposed to imposing syntactic formats), (2) capturing systems of interest without modifications, and (3) accommodating infinitary syntax and non-equivariant predicates.

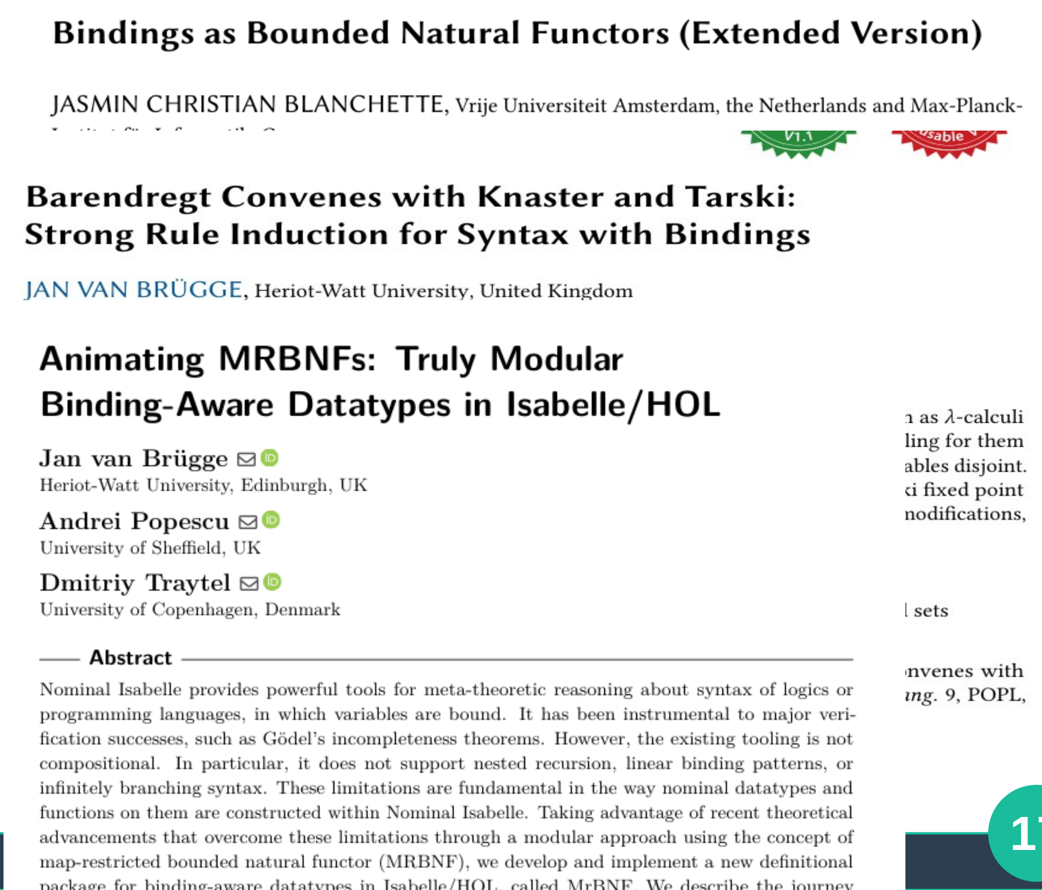
CCS Concepts: • **Theory of computation** → **Logic and verification**.

Additional Key Words and Phrases: syntax with bindings, induction, formal reasoning, nominal sets

**ACM Reference Format:**
Jan van Brügge, James McKinna, Andrei Popescu, and Dmitriy Traytel. 2025. Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings. *Proc. ACM Program. Lang.* 9, POPL, Article 57 (January 2025), 32 pages. https://doi.org/10.1145/3704893

## Short note on history

- POPL19 paper by Blanchette et. al. introducing MRBNFs
- POPL25 paper on strong rule induction for inductive predicates
- this paper showing binder_datatypes based on MRBNFs



**Bindings as Bounded Natural Functors (Extended Version)**

JASMIN CHRISTIAN BLANCHETTE, Vrije Universiteit Amsterdam, the Netherlands and Max-Planck-

**Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings**

JAN VAN BRÜGGE, Heriot-Watt University, United Kingdom

ı as $\lambda$-calculi
ling for them
ables disjoint.
ɩi fixed point
nodifications,

ı sets

nvenes with
ıng. 9, POPL,

**Animating MRBNFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL**

Jan van Brügge
Heriot-Watt University, Edinburgh, UK

Andrei Popescu
University of Sheffield, UK

Dmitriy Traytel
University of Copenhagen, Denmark

**Abstract**

Nominal Isabelle provides powerful tools for meta-theoretic reasoning about syntax of logics or programming languages, in which variables are bound. It has been instrumental to major verification successes, such as Gödel's incompleteness theorems. However, the existing tooling is not compositional. In particular, it does not support nested recursion, linear binding patterns, or infinitely branching syntax. These limitations are fundamental in the way nominal datatypes and functions on them are constructed within Nominal Isabelle. Taking advantage of recent theoretical advancements that overcome these limitations through a modular approach using the concept of map-restricted bounded natural functor (MRBNF), we develop and implement a new definitional package for binding-aware datatypes in Isabelle/HOL, called MrBNF. We describe the journey

## How to try it

## How to try it

1. Clone the repository
   https://github.com/jvanbruegge/binder_datatypes

## How to try it

1. Clone the repository
   https://github.com/jvanbruegge/binder_datatypes
2. Make Isabelle aware of the library:
   ```
   $ isabelle components -u /path/to/repo
   ```

## How to try it

1. Clone the repository
   https://github.com/jvanbruegge/binder_datatypes
2. Make Isabelle aware of the library:
   ```
   $ isabelle components -u /path/to/repo
   ```
3. import Binders.MRBNF_Recursor

## How to try it

1. Clone the repository
   https://github.com/jvanbruegge/binder_datatypes
2. Make Isabelle aware of the library:
   ```
   $ isabelle components -u /path/to/repo
   ```
3. import Binders.MRBNF_Recursor
4. Have fun :)

Also contains case studies and detailed example proofs for the automation

## Thanks

Questions