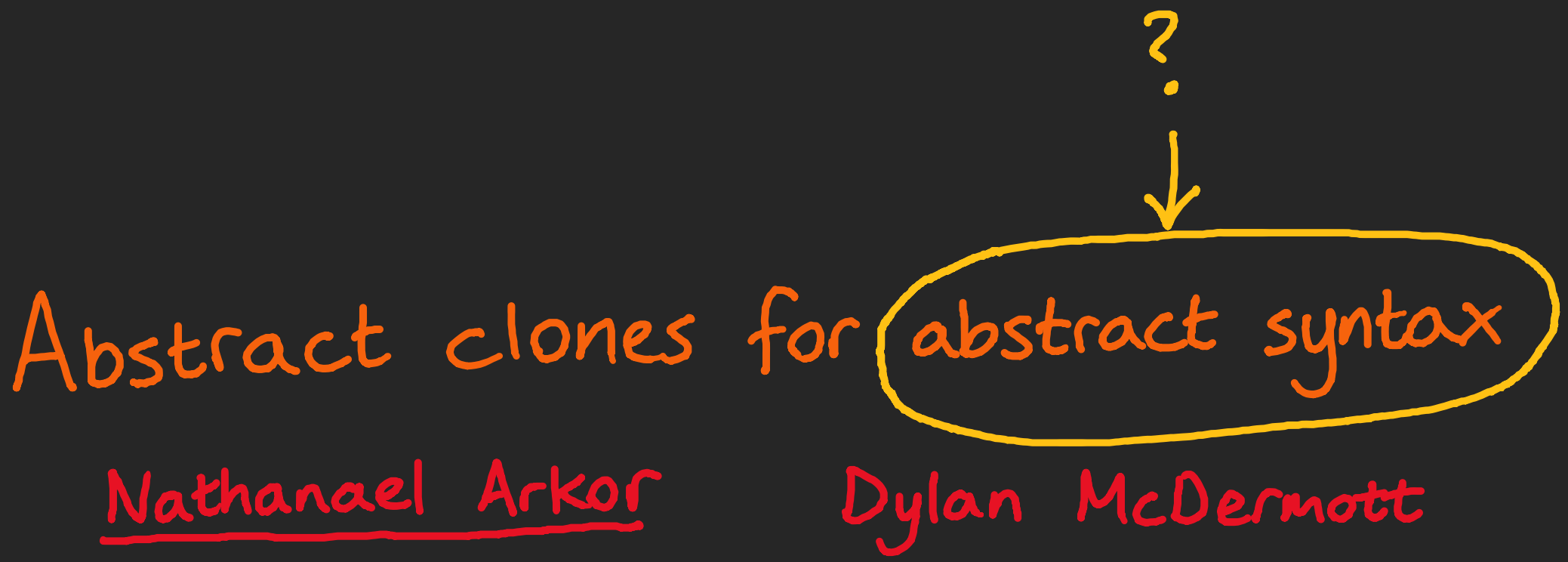
Abstract clones for abstract syntax

Nathanael Arkor Dylan McDermott

(FSCD 2021)



Nathanael Arkor Dylan McDermott

(FSCD 2021)

We want to be able to describe and reason about programming languages...

We want to be able to describe and reason about programming languages...

... but we don't want to reason about their concrete syntax, which is not a precise reflection of their abstract structure.

We want to be able to describe and reason about programming languages...

... but we don't want to reason about their concrete syntax, which is not a precise reflection of their abstract structure.

Abstract syntax is the name for the formalisms that capture the abstract structure of programming languages.

We want to be able to describe and reason about programming languages...

... but we don't want to reason about their concrete syntax, which is not a precise reflection of their abstract structure.

Abstract syntax is the name for the formalisms that capture the abstract structure of programming languages.

?

For us, a programming language will be a simple type theory, consisting of

- a collection of types;

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;
- equational laws on terms.

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;
- equational laws on terms.

equational theory (i.e. universal algebra)

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;
- equational laws on terms.

multisorted equational theory (i.e. multisorted universal algebra)

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;
- equational laws on terms.

multisorted equational theory + variable binding

For us, a programming language will be a simple type theory, consisting of

- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;
- equational laws on terms.

More precisely, this might be called a multisorted second-order equational theory (Fiore and Hur, 2010).

For us, a programming language will be a simple type theory, consisting of

? {
- a collection of types;
- a collection of (typed) terms in context;
- (capture-avoiding) simultaneous substitution;
- (variable-binding) operations on terms;
- equational laws on terms.

More precisely, this might be called a multisorted second-order equational theory (Fiore and Hur, 2010).

There exist formalisms in the literature to capture this structure.

- Σ-monoids of Fiore-Plotkin-Turi '99.
- Second-order algebraic theories of Fiore-Mahmoud '10.
- Structured monads and relative monads, cf. Ahrens, Hirschowitz, Hirschowitz, Lafont, Maggesi.
- Structured cartesian multicategories, à la Arkor-Fiore '20.

There exist formalisms in the literature to capture this structure.

(*)
- Σ-monoids of Fiore-Plotkin-Turi '99.
- Second-order algebraic theories of Fiore-Mahmoud '10.
- Structured monads and relative monads, cf. Ahrens, Hirschowitz, Hirschowitz, Lafont, Maggesi.
- Structured cartesian multicategories, à la Arkor-Fiore '20.

(*) Highly categorical

Is there an approach to the abstract syntax of simple type theories that does not require a high level of category-theoretic sophistication, yet is powerful enough to prove results we care about?

Is there an approach to the abstract syntax of simple type theories that does not require a high level of category-theoretic sophistication, yet is powerful enough to prove results we care about?

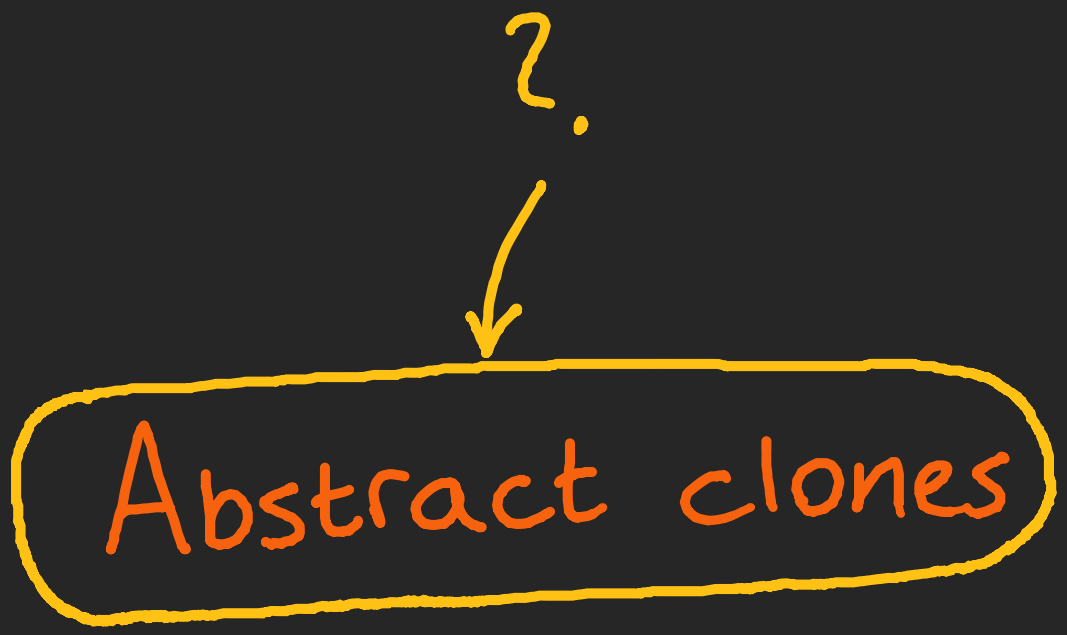
Yes.

Is there an approach to the abstract syntax of simple type theories that does not require a high level of category-theoretic sophistication, yet is powerful enough to prove results we care about?

Yes... and in fact this approach has appeared in limited cases in the literature previously (Mahmoud '11, Hyland '17). However, no unified account exists.

Is there an approach to the abstract syntax of simple type theories that does not require a high level of category-theoretic sophistication, yet is powerful enough to prove results we care about?

Yes... and in fact this approach has appeared in limited cases in the literature previously (Mahmoud '11, Hyland '17). However, no unified account exists. This is the purpose of our paper.



for abstract syntax

Nathanael Arkor Dylan McDermott

(FSCD 2021)

Abstract clones were introduced in 1965 as a way to axiomatize universal algebraic structure.

Abstract clones were introduced in 1965 as a way to axiomatize universal algebraic structure. Each specifies

- a collection of terms in context;
- simultaneous substitution operations.

Abstract clones were introduced in 1965 as a way to axiomatize universal algebraic structure. Each specifies

- a collection of terms in context;
- simultaneous substitution operations.

We can equivalently consider a term

$$x_1, \ldots, x_n \vdash t$$

as an operation

$$t : X^n \longrightarrow X$$

and in this way abstract clones capture algebra.

Abstract clones are equivalent to many other concepts (from category theory), for instance:

- Algebraic theories
- Cartesian operads
- Finitary monads on Set
- $(\mathrm{FinSet} \hookrightarrow \mathrm{Set})$-relative monads
- Substitution algebras of Fiore-Plotkin-Turi '99
- Monoids for the substitution tensor product

which are all equivalent to the traditional concept of equational theory.

Abstract clones

- An abstract clone X consists of
  - a set of terms for each context and type

$$X(\Gamma; B)$$

Abstract clones

- An abstract clone X consists of
  - a set of terms for each context and type
    $$X(\Gamma; B)$$
  - a family of elements for each context representing variable projections
    $$x_i \in X(x_1 : A_1, \ldots, x_n : A_n ; A_i) \quad (i \leq n)$$

Abstract clones

- An abstract clone X consists of
  - a set of terms for each context and type
    $$X(\Gamma ; B)$$
  - a family of elements for each context representing variable projections
    $$x_i \in X(x_1 : A_1 , \ldots , x_n : A_n \; ; \; A_i) \quad (i \leq n)$$
  - a function representing simultaneous substitution
    $$\frac{x_1 : A_1 , \ldots , x_n : A_n \vdash t : B \qquad \Gamma \vdash u_1 : A_1 \quad \cdots \quad \Gamma \vdash u_n : A_n}{\Gamma \vdash t \, [x_1 \mapsto u_1 , \ldots , x_n \mapsto u_n]}$$

satisfying laws axiomatizing substitution.

How do abstract clones capture algebraic structure? Consider the following two presentations of a binary operator:

$$\frac{}{x, y \vdash f(x, y)} \qquad \frac{\Gamma \vdash s \qquad \Gamma \vdash t}{\Gamma \vdash f(s, t)}$$

How do abstract clones capture algebraic structure?

Consider the following two presentations of a binary operator:

$$\frac{}{x, y \vdash f(x, y)}$$

operators as context-parameterized terms

$$\frac{\Gamma \vdash s \qquad \Gamma \vdash t}{\Gamma \vdash f(s, t)}$$

operators as context-indexed families of inference rules

How do abstract clones capture algebraic structure?

Consider the following two presentations of a binary operator:

$$\frac{}{x, y \vdash f(x, y)}$$

operators as context-parameterized terms

$$f \in X(*, * \,;\, *)$$

$$\frac{\Gamma \vdash s \qquad \Gamma \vdash t}{\Gamma \vdash f(s, t)}$$

operators as context-indexed families of inference rules

$$f : X(\Gamma;\, *)^2 \to X(\Gamma;\, *) \qquad (\forall \Gamma)$$

How do abstract clones capture algebraic structure?

Consider the following two presentations of a binary operator:

$$\frac{}{x, y \vdash f(x, y)} \qquad \sim \qquad \frac{\Gamma \vdash s \qquad \Gamma \vdash t}{\Gamma \vdash f(s, t)}$$

operators as context-parameterized terms

operators as context-indexed families of inference rules

$$f \in X(*, * \,;\, *) \qquad f : X(\Gamma;\, *)^2 \to X(\Gamma;\, *) \quad (\forall \Gamma)$$

Abstract clones capture algebraic structure, but we're interested in more than that. In particular, we want to capture variable-binding structure.

Abstract clones capture algebraic structure, but we're interested in more than that. In particular, we want to capture variable-binding structure.

Take the unityped λ-calculus, which is presented by the following rules (along with equations):

$$\frac{\Gamma \vdash f \quad \Gamma \vdash a}{\Gamma \vdash f\ a}\ (\text{app}) \qquad \frac{\Gamma, x \vdash t}{\Gamma \vdash \lambda x.\, t}\ (\text{abs})$$

Abstract clones capture algebraic structure, but we're interested in more than that. In particular, we want to capture variable-binding structure.

Take the unityped λ-calculus, which is presented by the following rules (along with equations):

$$\frac{\Gamma \vdash f \quad \Gamma \vdash a}{\Gamma \vdash f\ a}\ (\text{app}) \qquad \frac{\Gamma, x \vdash t}{\Gamma \vdash \lambda x.\, t}\ (\text{abs})$$

The terms of the unityped λ-calculus form an abstract clone Λ.

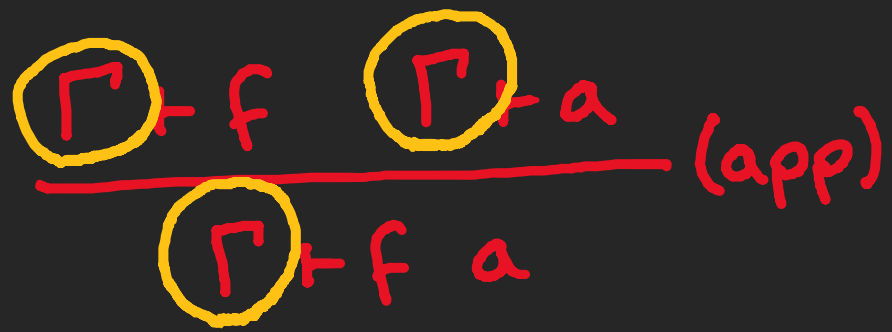
$$\frac{\Gamma \vdash f \quad \Gamma \vdash a}{\Gamma \vdash f\ a}\,(app)$$

Application is an algebraic operation and is therefore captured by the clone structure, as an element

$$app \in \Lambda(*, * \,;\, *)$$

which induces a family of functions

$$[\![app]\!] : \Lambda(\Gamma\,;\, *) \times \Lambda(\Gamma\,;\, *) \longrightarrow \Lambda(\Gamma\,;\, *) \qquad (\forall \Gamma)$$



Application is an algebraic operation and is therefore captured by the clone structure, as an element

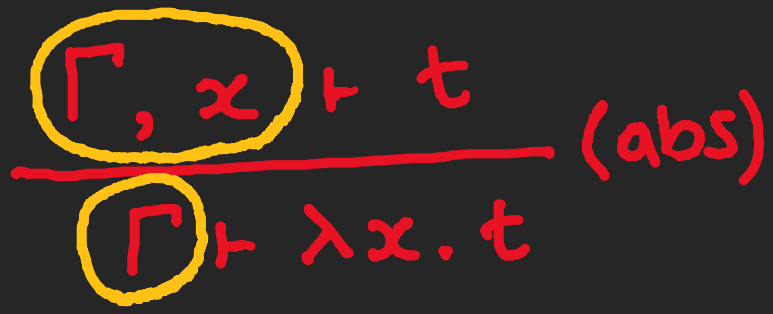
$$app \in \Lambda(*, * \,;\, *)$$

which induces a family of functions

$$[\![app]\!] : \Lambda(\Gamma\,;\,*) \times \Lambda(\Gamma\,;\,*) \longrightarrow \Lambda(\Gamma\,;\,*) \quad (\forall \Gamma)$$

$$\frac{\Gamma, x \vdash t}{\Gamma \vdash \lambda x.\, t}\ (\text{abs})$$

However, abstraction is not an algebraic operation, since it involves a variable binding. Therefore, it is not captured by the clone structure.



However, abstraction is not an algebraic operation, since it involves a variable binding. Therefore, it is not captured by the clone structure.

$$\frac{\Gamma, x \vdash t}{\Gamma \vdash \lambda x.\, t}\ (\text{abs})$$

However, abstraction is not an algebraic operation, since it involves a variable binding. Therefore, it is not captured by the clone structure.

Observe that, if the abstraction was captured by the abstract clone, it would induce a family of functions

$$[\![\text{abs}]\!] : \Lambda(\Gamma, * \,;\, *) \longrightarrow \Lambda(\Gamma \,;\, *) \qquad (\forall \Gamma)$$

In fact, it suffices to axiomatize this derived structure, to capture variable binding.

Idea

Universal algebraic structure is captured by algebraic structure on sets:

$$M \times M \xrightarrow{\otimes} M$$

(+ equations)

Variable-binding structure is captured by algebraic structure on abstract clones:

$$\Lambda(\Gamma, * \,;\, *) \xrightarrow{\text{abs}} \Lambda(\Gamma \,;\, *) \qquad (\forall \Gamma)$$

(+ equations)

Idea

Universal algebraic structure is captured by algebraic structure on sets:

$$M \times M \xrightarrow{\otimes} M$$

(+ equations)

Variable-binding structure is captured by algebraic structure on abstract clones:

$$\Lambda(\Gamma, *; *) \xrightarrow{\text{abs}} \Lambda(\Gamma; *) \qquad (\forall \Gamma)$$

(+ equations)

need not be homogeneous

Therefore, to capture simple type theories, we may consider abstract clones equipped with algebraic structure. This leads to a conceptually elegant and practical approach to abstract syntax that avoids the categorically sophisticated constructions arising in other approaches.

Therefore, to capture simple type theories, we may consider abstract clones equipped with algebraic structure. This leads to a conceptually elegant and practical approach to abstract syntax that avoids the categorically sophisticated constructions arising in other approaches.

What does this look like, practically?

A simple type theory is usually presented by means of natural deduction rules (formation rules, introduction rules, elimination rules, computation rules, etc.). Formally, this corresponds to a second-order presentation:

$$(\Sigma, E)$$

Σ: signature of variable-binding operators

E: equations on derived terms

A simple type theory is usually presented by means of natural deduction rules (formation rules, introduction rules, elimination rules, computation rules, etc.). Formally, this corresponds to a second-order presentation:

$$(\Sigma, E)$$

Σ: signature of variable-binding operators

E: equations on derived terms

The algebras for a presentation are abstract clones equipped with algebraic structure corresponding to the rules described by the presentation.

Algebras for a second-order presentation are the simple type theories interpreting the rules of the presentation. We are usually interested in a canonical algebra: the one generated freely from the rules. This is the free algebra for the presentation, and may be understood as the abstract syntax for our programming language.

Algebras for a second-order presentation are the simple type theories interpreting the rules of the presentation. We are usually interested in a canonical algebra: the one generated freely from the rules. This is the free algebra for the presentation, and may be understood as the abstract syntax for our programming language.

Every presentation admits a free algebra, given by an inductive construction using the second-order equational logic of Fiore and Hur.

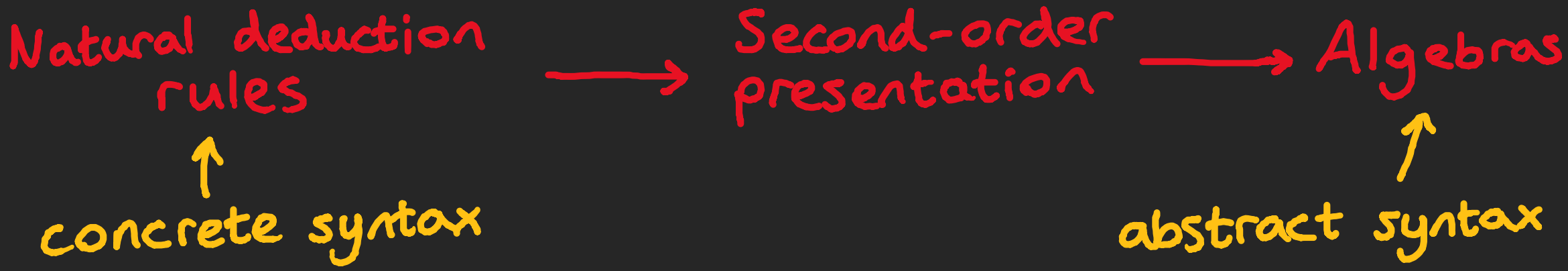
Having an abstract representation of a type theory is only useful if it facilitates the proofs of interesting theorems.

Having an abstract representation of a type theory is only useful if it facilitates the proofs of interesting theorems.

We show how abstract clones with algebraic structure can be used to perform logical relations arguments. In particular, we prove

- adequacy for the set-theoretic model of the STLC
- normalization for the STLC
- ... with global state

High-level overview



We may then prove theorems about the programming language using more abstract and elegant techniques, facilitated by general metatheorems, such as our induction principle for second-order syntax.

Thesis

Abstract clones provide a viable and practical approach to abstract syntax especially well-suited for applications in which category-theoretic techniques are avoided.

Thesis

Abstract clones provide a viable and practical approach to abstract syntax especially well-suited for applications in which category-theoretic techniques are avoided.

For the complete framework, more details, and applications, see the paper:

Abstract clones for abstract syntax

Thanks for listening!