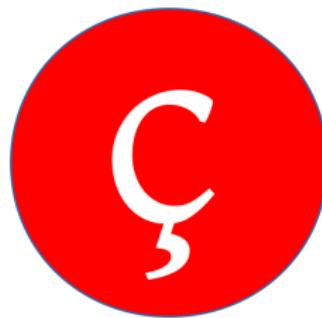


# Introduction to Programming and Proving in Cedille



Chris Jenkins, Colin McDonald, Aaron Stump  
Computer Science  
The University of Iowa  
Iowa City, Iowa

# Plan for the tutorial

ζ ?

↪ CeDiLIE

cedille

~> cedille<sub>core</sub>

c d ll



# Plan for the tutorial

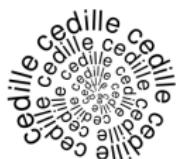


↪ *CeDiLIE*

cedille

~> cedille<sub>core</sub>

c d ll



Motivation and background for Cedille

Syntax and semantics

Tooling: emacs frontend ↔ backend

Elaboration to Cedille Core

Spine-local type inference

Future directions



Motivation and background for Cedille

## A little history

# A little history



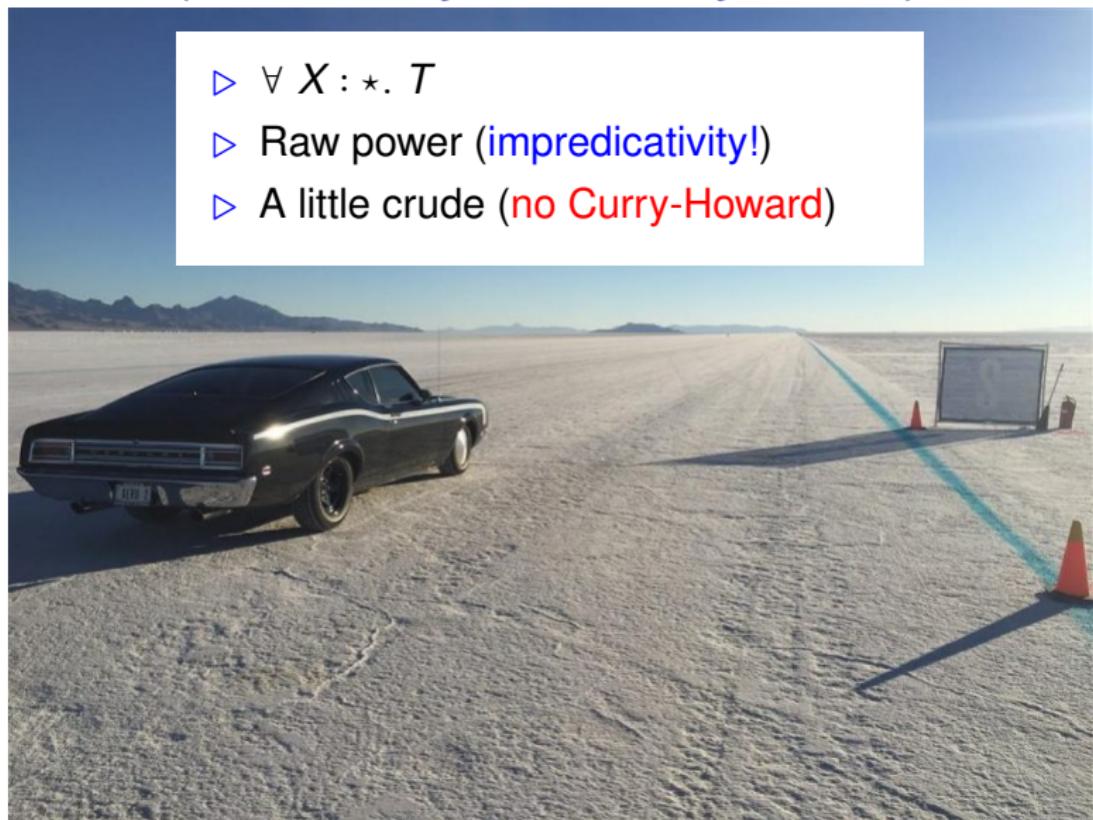
## System F (Girard, Reynolds, early 1970s)



*1969 Mercury Cyclone Spoiler II*

## System F (Girard, Reynolds, early 1970s)

- ▷  $\forall X : \star. T$
- ▷ Raw power (**impredicativity!**)
- ▷ A little crude (**no Curry-Howard**)



1969 *Mercury Cyclone Spoiler II*

# Calculus of Constructions (Coquand, Huet 1988)



*1988 Chevrolet Camaro*

# Calculus of Constructions (Coquand, Huet 1988)

- ▷ Add dependent types:  $\Pi x : T. T'$
- ▷ Imported from Automath/Martin-Löf type theory
- ▷ Curry-Howard!
- ▷ No induction. [Geuvers 2001]



1988 Chevrolet Camaro

# Calculus of Inductive Constructions (Werner 1994)



1992 Hoffman-Markley Streamliner

# Calculus of Inductive Constructions (Werner 1994)

- ▷ Add primitive inductive types
- ▷ Finally ready for constructive mathematics!
- ▷ Basis for Coq



1992 Hoffman-Markley Streamliner

## But Coq $\neq$ CIC

- ▷ Coinductive types
- ▷ Universe hierarchy (Extended CC, Luo 1990)
- ▷ Proof-irrelevant universe  $\text{Prop}$
- ▷ *And we might want more:*
  - definitional proof irrelevance
  - inductive-inductive types
  - inductive-recursive types

Similarly, Agda  $\neq$  MLTT.

# Issues and limitations, Coq and Agda

- ▷ No formal semantics/correctness proof
  - ▶ Despite a lot of interest: TT in TT
- ▷ (Hence!) bugs and surprises
  - ▷ incompatibilities with various axioms
  - ▷ actual contradictions!
  - ▷ type soundness broken in Coq
- ▷ Commitment to a set of datatypes
  - ▷ theory of datatypes not finished...
  - ▷ e.g., higher-order abstract syntax prohibited

# Have we created a monster?



Schaufelradbagger 258

# Have we created a monster?



*Schaufelradbagger 258*

## *If I could turn back time...*

Good-bye to:

- ▷ primitive datatypes
- ▷ (also universe hierarchy, my bias)

Hello to

- ▷ lambda-encodings of data

## *If I could turn back time...*

Good-bye to:

- ▷ primitive datatypes
- ▷ (also universe hierarchy, my bias)

Hello to

- ▷ lambda-encodings of data



## *If I could turn back time...*

Good-bye to:

- ▷ primitive datatypes
- ▷ (also universe hierarchy, my bias)

Hello to

- ▷ lambda-encodings of data

# Wanted: a new type theory

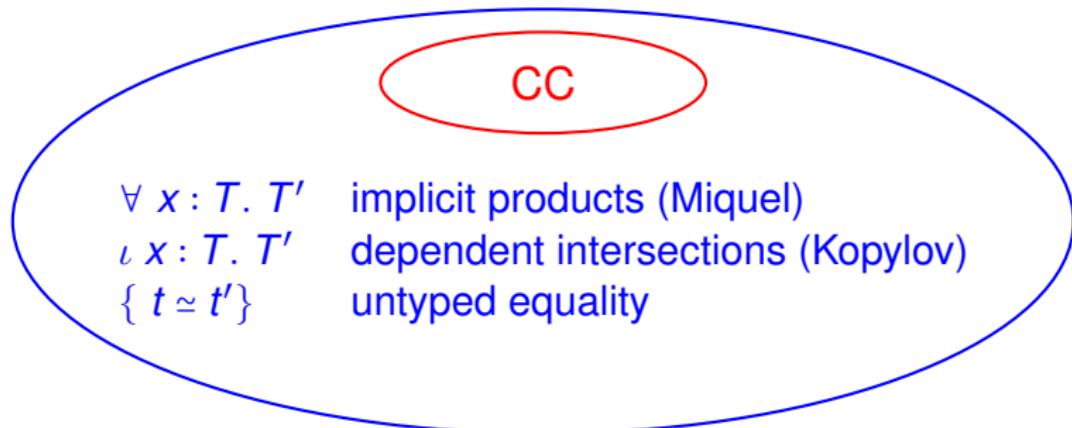
where

- ▷ inductive datatypes are derived (lambda-encoded)
- ▷ impredicativity is central
- ▷ core theory is small and verifiable

Tooling goals:

- ▷ see all typing/inference information
- ▷ predictable inference
- ▷ elaborate to core with independent checker

# Introducing Cedille



- ▷ Small theory, formal syntax and semantics
- ▷ Core checker implemented in < 1000loc Haskell
- ▷ Logically sound
- ▷ Turing complete(!)
- ▷ Supports inductive lambda-encodings

Back the truck up

# Back the truck up

Did you say lambda encodings?



## Not your forebear's lambda encodings

- ▷ Usual rap: inefficient accessors
- ▷ Corrected by Parigot 1988 for typed encoding
- ▷ Perfect untyped encoding Böhm et al. 1994
  - linear space
  - constant-time accessors
  - intrinsic support for iteration
- ▷ Cedille: perfect inductive (typed) encodings

What do we get from this?



What do we get from this?

*Freedom*



# What do we get from this?

## *Freedom*

- ▷ No pre-set datatype class
- ▷ Explore semantics of advanced datatypes
- ▷ *Power of impredicativity*
- ▷ So far: Functorial, Monotone, IR, II

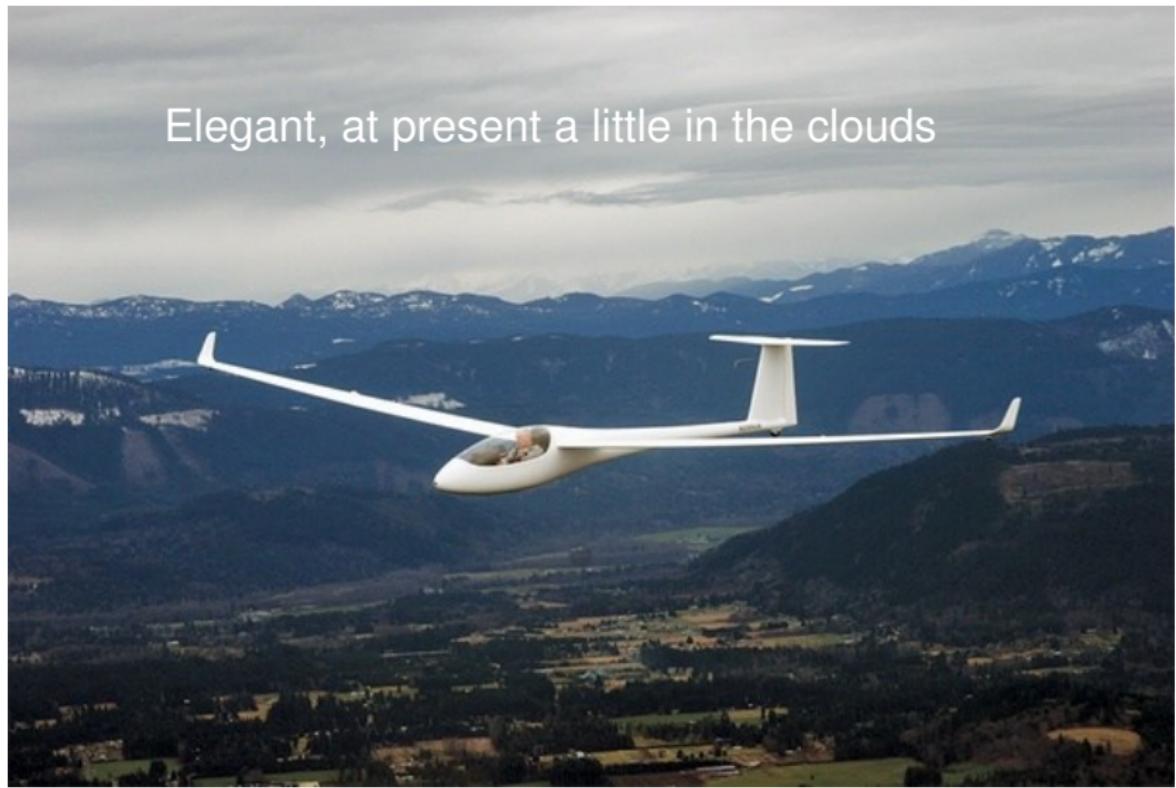
So which car are we?

So which car are we?



# So which car are we?

Elegant, at present a little in the clouds



$\vdash$  *CeDiLIE*

Syntax and semantics of Cedille

Calculus of Dependent Lambda Eliminations

CC

 $\forall x : T. \ T'$   
 $\iota x : T. \ T'$   
 $\{ t \simeq t' \}$ 

implicit products (Miquel)  
dependent intersections (Kopylov)  
untyped equality

## Dependent intersections $\iota x : T_1. T_2$

- ▷ Usual intersection types:

$$\frac{t : T_1 \quad t : T_2}{t : T_1 \cap T_2}$$

- ▷ Dependent intersection:

$$\frac{t : T_1 \quad t : [t/x]T_2}{t : \iota x : T_1. T_2}$$

$T_2$  can refer to subject of typing, at weaker type  $T_1$

But if you are using intersections... 

But if you are using intersections... 

You must have an **extrinsic** (Curry-style) type theory.

But if you are using intersections... 

You must have an **extrinsic** (Curry-style) type theory.

- ▷ Unannotated terms of pure lambda calculus

## But if you are using intersections...

You must have an **extrinsic** (Curry-style) type theory.

- ▷ Unannotated terms of pure lambda calculus
- ▷ Assign multiple different types to same term  
(Intrinsic type theories usually have unique types.)

## But if you are using intersections...

You must have an **extrinsic** (Curry-style) type theory.

- ▷ Unannotated terms of pure lambda calculus
- ▷ Assign multiple different types to same term  
(Intrinsic type theories usually have unique types.)
- ▷ Completely different from Coq, Agda

## But if you are using intersections...

You must have an **extrinsic** (Curry-style) type theory.

- ▷ Unannotated terms of pure lambda calculus
- ▷ Assign multiple different types to same term  
(Intrinsic type theories usually have unique types.)
- ▷ Completely different from Coq, Agda
- ▷ Much less explored TT...

## But if you are using intersections...

You must have an **extrinsic** (Curry-style) type theory.

- ▷ Unannotated terms of pure lambda calculus
- ▷ Assign multiple different types to same term  
(Intrinsic type theories usually have unique types.)
- ▷ Completely different from Coq, Agda
- ▷ Much less explored TT...

Yes!

## For each type (here, implicit products)

- Formation rule for the type

$$\frac{\Gamma \vdash T' : * \quad \Gamma, x : T' \vdash T : *}{\Gamma \vdash \forall x : T'. T : *}$$

- Introduction and elimination rules

$$\frac{\Gamma \vdash \forall x : T'. T : * \quad \Gamma, x : T' \vdash t : T}{\Gamma \vdash t : \forall x : T'. T}$$
    
$$\frac{\Gamma \vdash t : \forall x : T'. T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t : [t'/x]T}$$

- Annotated terms

- Introduction:  $\Lambda x : T'. t$
- Elimination:  $t - t'$

- Annotated terms erase to pure lambda terms

$$|\Lambda x : T. t| = |t|$$

$$|t - t'| = |t|$$

# Equality type

Formation:

$$\frac{FV(t'') \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \{t \simeq t'\} : *}$$

Introduction and elimination:

$$\frac{\begin{array}{c} FV(t'') \subseteq \text{dom}(\Gamma) \\ \Gamma \vdash t' : \{t_1 \simeq t_2\} \end{array}}{\Gamma \vdash t : \{t' \simeq t'\}}$$
    
$$\frac{\begin{array}{c} \Gamma \vdash t : [t_1/x]T \\ \Gamma \vdash t : [t_2/x]T \end{array}}{\Gamma \vdash t : [t_2/x]T}$$

Direct computation rule:

$$\frac{\begin{array}{c} \Gamma \vdash t : \{t_1 \simeq t_2\} \\ \Gamma \vdash t_1 : T \end{array}}{\Gamma \vdash t_2 : T}$$

Annotations:

$$\begin{aligned} |\beta\{t\}| &= |t| \\ |\rho t' - t| &= |t| \\ |\phi t - t_1\{t_2\}| &= |t_2| \end{aligned}$$

## The Kleene trick

- Any term can be assigned a trivially true equality

$$\textcolor{red}{t} : \{t' \simeq t'\}$$

- Restricted form of subset type, combined with *conversion*

$$\frac{\Gamma \vdash t : T' \quad \Gamma \vdash T : * \quad T \cong T'}{\Gamma \vdash t : T}$$

- When  $[t/x]t' =_{\beta\eta} [t/x]t''$ , yields

$$t : \textcolor{red}{\lambda x : T. \{t' \simeq t''\}}$$

- E.g.,

$$\textit{True} = \forall X : *. X \rightarrow X$$

$$\lambda y. y y : \textcolor{red}{\lambda x : \textit{True} \rightarrow \textit{True}. \{x \lambda z. z \simeq \lambda z. z\}}$$

$$\textit{anything} : \{\lambda x. x \simeq \lambda x. x\}$$

## Aside on rewriting

$$\rho \ t - t'$$

- ▷ Suppose  $t : \{t_1 \simeq t_2\}$ , and  $T$  is type for  $t'$ .
- ▷ Find each subterm of  $T$  convertible to  $t_1$  and rewrite.
- ▷ Error if no matches
- ▷ Use  $\rho$  anywhere in a term (cf. Agda)
- ▷ Other forms of  $\rho$ :
  - $\rho+$ , head-normalize as you look for matches
  - $\rho(n_1 \dots n_k)$ , rewrite occurrences  $n_1, \dots n_k$

# Dependent intersections

Formation:

$$\frac{\Gamma \vdash T : * \quad \Gamma, x : T \vdash T' : *}{\Gamma \vdash \iota x : T. T'}$$

Introduction and elimination:

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash \textcolor{blue}{t} : [t/x]T'}{\Gamma \vdash t : \iota x : T. T'} \quad \frac{\Gamma \vdash t : \iota x : T. T' \quad \Gamma \vdash t : \iota x : T. T'}{\Gamma \vdash t : T} \quad \frac{\Gamma \vdash t : \iota x : T. T'}{\Gamma \vdash t : [t/x]T'}$$

Annotations:

$$|[t, \textcolor{blue}{t'}]| = |t|$$

$$|t.1| = |t|$$

$$|t.2| = |t|$$

# How are inductive datatypes defined?

- ▷ Several variations (CPP '18, ITP '18), one theme:

The type of  $d$  expresses an induction principle for  $d$

- ▷ For Nat:

$$n : \forall P : Nat \rightarrow *. (\forall x : Nat. P x \rightarrow P (S x)) \rightarrow P Z \rightarrow P n$$

- ▷ Essentially due to Leivant 1983
- ▷ Will walk through example a little later
- ▷ With D. Firsov, generic derivations for classes of  $F : * \rightarrow *$

## Another example: casts

- ▷ A cast is an identity function from A to B

Cast :  $\star \rightarrow \star \rightarrow \star = \lambda A : \star . \lambda B : \star .$   
 $\iota \text{ cast} : A \rightarrow B . \{ \text{cast} \simeq \lambda x . x \}.$

- ▷ If there is a cast, you can change the type:

cast :  $\forall A : \star . \forall B : \star . \text{Cast} \cdot A \cdot B \Rightarrow A \rightarrow B$   
= ...

- ▷ Nontrivial casts exist in extrinsic type theory:

$\lambda x. x : (\forall X : \star. X \rightarrow X) \rightarrow (Nat \rightarrow Nat)$

- ▷ Not in intrinsic type theory:

$\lambda x. (x \cdot Nat) : (\forall X : \star. X \rightarrow X) \rightarrow (Nat \rightarrow Nat)$

## Monotone recursive types

- ▷ Explored in works by R. Matthes [eg, in *Synthese*, 2002]
- ▷ Given monotone  $F$ , extend the theory with  $\mu F$
- ▷ Monotonicity expressed by a term  $t$  of type
$$\forall X:\star. \forall Y:\star. (X \rightarrow Y) \rightarrow (F \cdot X \rightarrow F \cdot Y)$$
- ▷ Matthes considers how to extend theory, retaining SN

## Monotone recursive types

- ▷ Explored in works by R. Matthes [eg, in *Synthese*, 2002]
- ▷ Given monotone  $F$ , extend the theory with  $\mu F$
- ▷ Monotonicity expressed by a term  $t$  of type
$$\forall X:\star. \forall Y:\star. (X \rightarrow Y) \rightarrow (F \cdot X \rightarrow F \cdot Y)$$
- ▷ Matthes considers how to extend theory, retaining SN

*We can derive a stronger form, within our theory!*

## Starting point: proof of Tarski's Theorem

Consider monotone  $f : \mathcal{P}(X) \rightarrow \mathcal{P}(X)$

Let  $Q = \bigcap\{A \mid f(A) \subseteq A\}$

Prove  $f(Q) = Q$  by both inclusions.

## Translating the proof to Cedille

- ▷ “ $A \subseteq B$ ” becomes `Cast · A · B`

- ▷ Monotonicity becomes

$\forall X : * . \forall Y : *$ .

`Cast · X · Y → Cast · (F · X) · (F · Y)`

- ▷ “ $\cap\{A \mid f(A) \subseteq A\}$ ” becomes

`Rec = ∀ X : * . Cast · (F · X) · X ⇒ X`

## Monotone recursive types: summary

- ▷ Derive  $\text{Rec}$  for any monotone  $F$
- ▷ Casts (identity functions) between  $F \dashv \text{Rec}$  and  $\text{Rec}$
- ▷ Proof translates proof of Tarski fixed-point theorem
- ▷ Code is 18 lines of Cedille
- ▷ To emphasize: no addition to the theory, these are derived

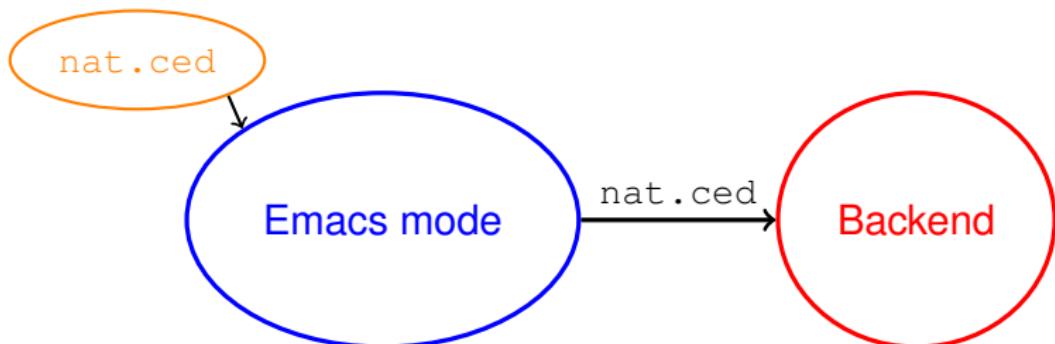
# cedille

Tooling: emacs frontend ↔ backend

[cedille.github.io](https://cedille.github.io)

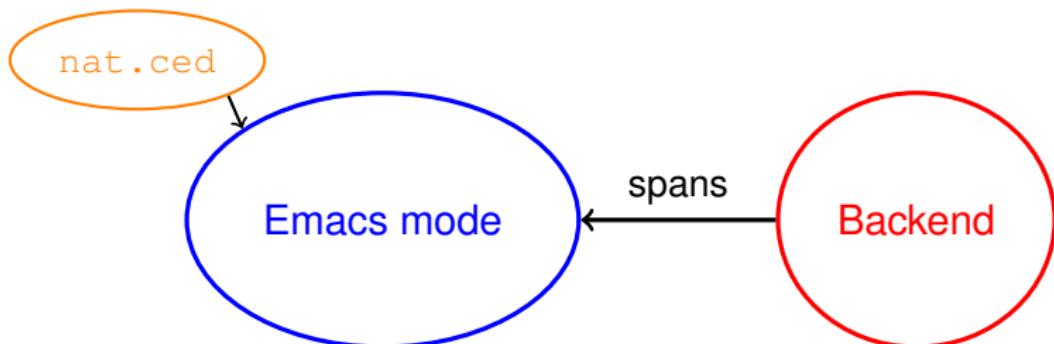
## Emacs mode

- ▶ Interact with Cedille through an emacs mode



## Emacs mode

- ▷ Interact with Cedille through an emacs mode



- ▷ A span is  $[label, start\text{-}pos, end\text{-}pos, attributes]$
- ▷ Spans communicated in JSON
- ▷ Cedille sends all type information, in span attributes
- ▷ Monadic style for writing the backend (type checker)

Demo cedille