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

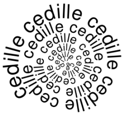
ç?

$\vdash$ *CeDilLE*

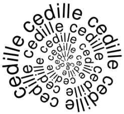
`cedille`

$\leadsto$ `cedille`$_{\texttt{core}}$

`c d ll`



Plan for the tutorial

Ç?	Motivation and background for Cedille
$\vdash$ *CeDilLE*	Syntax and semantics
`cedille`	Tooling: emacs frontend $\leftrightarrow$ backend
$\leadsto$ $\texttt{cedille}_{\texttt{core}}$	Elaboration to Cedille Core
`c d ll`	Spine-local type inference
	Future directions

Ç?

Motivation and background for Cedille

A little history

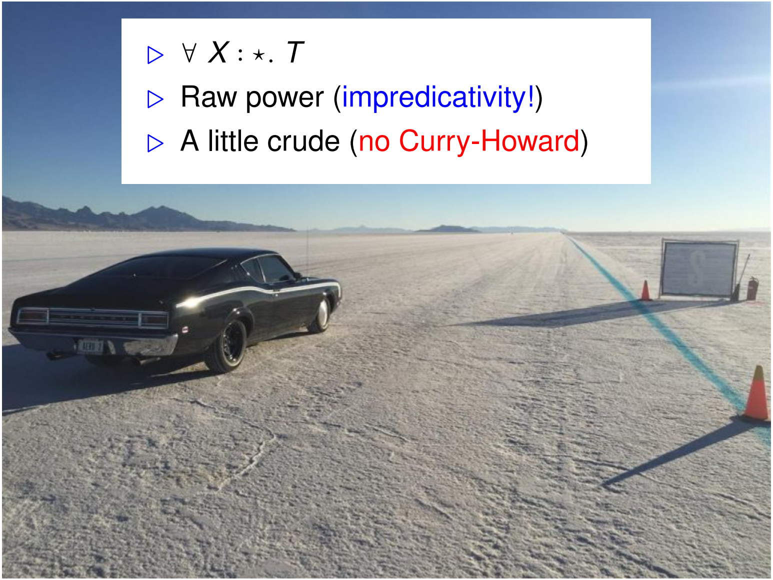
A little history

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)



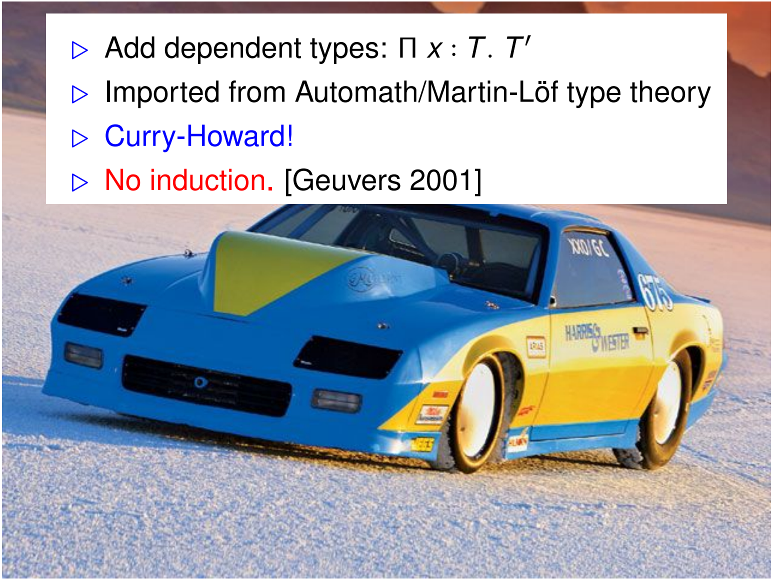
1969 Mercury Cyclone Spoiler II

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 `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 <u>derived</u> (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

CC

$\forall\, x : T.\ T'$	implicit products (Miquel)
$\iota\, x : T.\ T'$	dependent intersections (Kopylov)
$\{\ t \simeq t' \}$	untyped equality

- 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

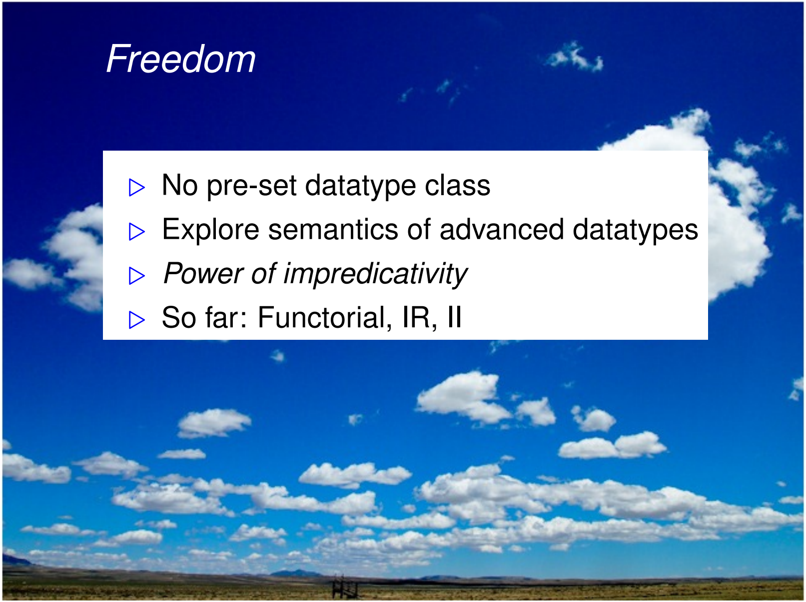
Not your forebear's lambda encodings

- 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?

What do we get from this?



So which car are we?

So which car are we?

So which car are we?

$\vdash$ *CeDilLE*

Syntax and semantics of Cedille

Types T and kinds κ

$\forall\, X : \kappa.\ T$	Impredicative quantification
$\lambda\, X : \kappa.\ T$	Type-level function with type input
$\lambda\, x : T.\ T'$	Type-level function with term input
$T\ T'$	Type-level application to a type
$T\ t$	Type-level application to a term
$\Pi\, x : T.\ T'$	Dependent function type
$\forall\, x : T.\ T'$	Implicit product
$\iota\, x : T.\ T'$	Dependent intersection
$\{\ t \simeq t'\ \}$	Untyped equality

$\star$	Kind for types
$\Pi\, x : T.\ \kappa$	Kind for predicates on terms
$\Pi\, X : \kappa.\ \kappa'$	Kind for predicates on types

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

- Usual intersection types:

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

- Dependent intersection:

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

$\boxed{T_2 \text{ can refer to subject of typing, at weaker type } T_1}$

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

- Formation rule for the type

$$\frac{\Gamma \vdash T' : \star \qquad \Gamma, x : T' \vdash T : \star}{\Gamma \vdash \forall\ x : T'.\ T : \star}$$

- Introduction and elimination rules

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

- Annotated terms
  - Introduction: $\Lambda\ x : T'.\ t$
  - Elimination: $t\ {-}t'$

- Annotated terms erase to pure lambda terms

$$\begin{array}{rcl} |\Lambda\ x : T.\ t| & = & |t| \\ |t\ \text{-}t'| & = & |t| \end{array}$$

Equality type

Formation:

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

Introduction and elimination:

$$\frac{FV(t') \subseteq dom(\Gamma)}{\Gamma \vdash \color{blue}{t} : \{t' \simeq t'\}} \quad \frac{\Gamma \vdash t' : \{t_1 \simeq t_2\} \qquad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash t : [t_2/x]T}$$

Direct computation rule:

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \qquad \Gamma \vdash t_1 : T}{\Gamma \vdash t_2 : T}$$

Annotations:

$$\begin{array}{rcl} |\beta\{\color{blue}{t}\}| & = & |t| \\ |\rho\ t'\ \text{-}\ t| & = & |t| \\ |\phi\ t\ \text{-}\ t_1\{t_2\}| & = & |t_2| \end{array}$$

Equality type

Formation:

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

Introduction and elimination:

$$\frac{FV(t') \subseteq dom(\Gamma)}{\Gamma \vdash t : \{t' \simeq t'\}} \quad \frac{\Gamma \vdash t' : \{t_1 \simeq t_2\} \qquad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash t : [t_2/x]T}$$

Direct computation rule:

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \qquad \Gamma \vdash t_1 : T}{\Gamma \vdash t_2 : T}$$

Annotations:

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

Dependent intersections

Formation:

$$\frac{\Gamma \vdash T : \star \qquad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \iota\, x : T.\ T'}$$

Introduction and elimination:

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

Annotations:

$$\begin{array}{rcl} |[t, t']| & = & |t| \\ |t.1| & = & |t| \\ |t.2| & = & |t| \end{array}$$

How are inductive datatypes defined?

- Many variations, one theme:

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

- For Nat:

$$n \ : \ \forall\ P : Nat \to \star.\ (\forall\ x : Nat.\ P\ x \to P\ (S\ x)) \to P\ Z \to P\ n$$

- Essentially due to Leivant 1983
- Will walk through example a little later

Another example: casts

- A cast is an identity function from A to B

```
Cast : ⋆ → ⋆ → ⋆ = λ A : ⋆ . λ B : ⋆ .
    ι cast : A → B . { cast ≃ λ x . x }.
```

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

```
cast : ∀ A : ⋆ . ∀ B : ⋆ . Cast · A · B ⇒ A → B
  = ...
```

- Nontrivial casts exist in extrinsic type theory:

$$\lambda\, x.\, x \quad : \quad (\forall\, X{:}\star.\, X \to X) \to (Nat \to Nat)$$

- Not in intrinsic type theory:

$$\lambda\, x.\, (x \cdot Nat) \quad : \quad (\forall\, X{:}\star.\, X \to X) \to (Nat \to Nat)$$

Monotone recursive types

- Explored in works by R. Matthes [eg, in Synthese, 2002]
- Given monotone F, extend the theory with μF
- Monotonicity expressed by a term t of type

  $\forall X{:}\star.\, \forall Y{:}\star.\, (X \to Y) \to (F \cdot X \to F \cdot Y)$
- Matthes considers how to extend theory, retaining SN

Monotone recursive types

- Explored in works by R. Matthes [eg, in Synthese, 2002]
- Given <u>monotone</u> F, extend the theory with μF
- Monotonicity expressed by a term t of type

  $\forall X{:}\star.\, \forall Y{:}\star.\, (X \to Y) \to (F \cdot X \to F \cdot Y)$
- 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 $f : \mathcal{P}(X) \to \mathcal{P}(X)$

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

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

- For $f(Q) \subseteq Q$, show $f(Q) \subseteq A$ for all $f(A) \subseteq A$
  - Assume A s.t. $f(A) \subseteq A$.
  - $Q \subseteq A$
  - $f(Q) \subseteq f(A)$ by mono.
  - $f(A) \subseteq A$
- Now by mono., $f(f(Q)) \subseteq f(Q)$, so $f(Q) \in \{A \mid f(A) \subseteq A\}$
  - Hence $Q \subseteq f(Q)$

Translating the proof to Cedille

- "$A \subseteq B$" becomes `Cast · A · B`
- Monotonicity becomes

```
∀ X : ★ . ∀ Y : ★ .
Cast · X · Y → Cast · (F · X) · (F · Y)
```

- "$\bigcap\{A \mid f(A) \subseteq A\}$" becomes

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

- "$f(Q) \subseteq Q$" becomes

```
Cast · (F · Rec)) · Rec
```

- similarly for "$Q \subseteq f(Q)$":

```
Cast · Rec · (F · Rec)
```

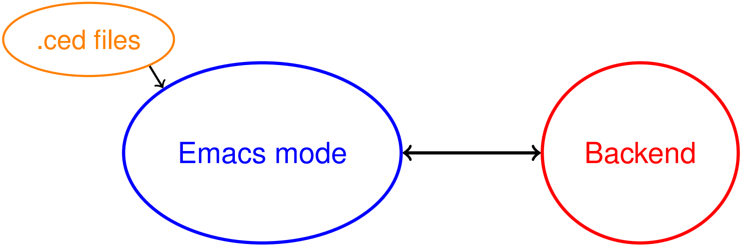
Monotone recursive types: summary

- Derive `Rec` for any montone `F`
- Casts (identity functions) between `F · Rec` and `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

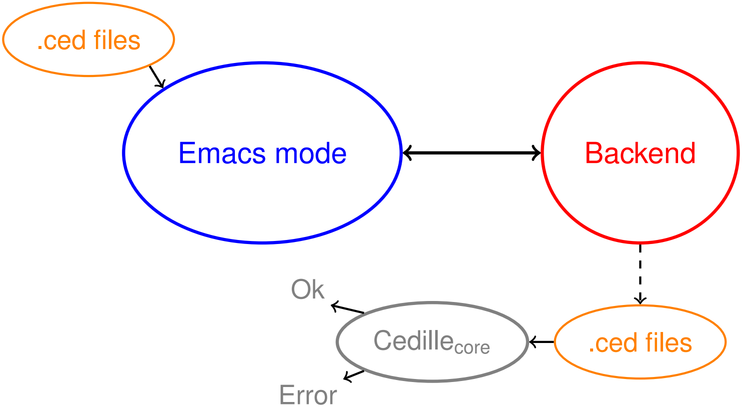
`cedille`

Tooling: emacs frontend $\leftrightarrow$ backend

Architecture of Cedille

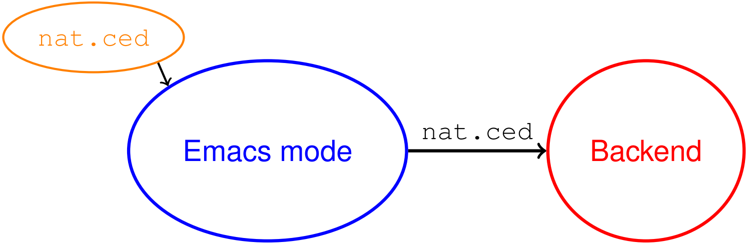


Architecture of Cedille



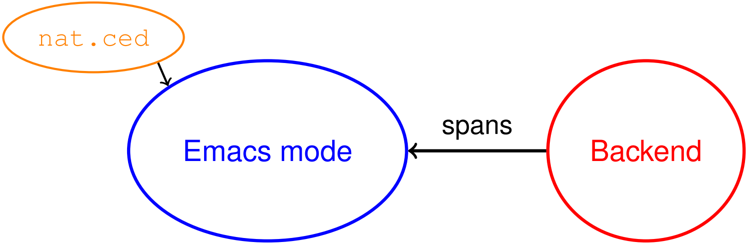
Emacs mode

- Interact with Cedille through an emacs mode



Emacs mode

- Interact with Cedille through an emacs mode



- A span is [*label*, *start-pos*, *end-pos*, *attributes*]
- Spans communicated in JSON
- Cedille sends <u>all</u> type information, in span attributes
- Monadic style for writing the backend (type checker)

Demo `cedille`