Rebuilding Constructive Type Theory with Cedille

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa

Rediscovering Constructive Type Theory with Cedille

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa

In this talk, let us

Seek out **green type theory**, by

going back to a simpler time,

without data,

only λ.

Plan for the talk

ç Cedille, motivation and architecture

merge Histomorphic mergesort

ç Current and future directions

Ç Cedille, motivation and architecture

The mission: seek out this greener type theory,

passing through perilous, unexplored territory

following the course of

The mission: seek out this greener type theory,

passing through perilous, unexplored territory

following the course of the Missouri River...?

The object of your mission is to explore the Missouri river, & such principal stream of it, as, by it's course & communication with the water of the Pacific ocean may offer the most direct & practicable water communication across this continent, for the purposes of commerce.

The object of your mission is to explore the Missouri river, & such principal stream of it, as, by it's course & communication with the water of the Pacific ocean may offer the most direct & practicable water communication across this continent, for the purposes of commerce.

Thomas Jefferson to Meriwether Lewis, June 20, 1803.

Let us form a type-theoretic Corps of Discovery!

Let us form a type-theoretic Corps of Discovery!

For a tour of the recent history of type theory


BRITISH TERRITORY
OREGON COUNTRY
Fort Clatsop, Oregon
Lewiston, Idaho
Great Falls, Montana
Billings, Montana
New Town, North Dakota
Circle of Cultures, North Dakota
ILLINOIS COUNTRY
MICH. CNTRY
Oacoma/Chamberlain, South Dakota
LOUISIANA PURCHASE
Omaha, Nebraska
Kansas City, Missouri
St. Charles
Hartford, Illinois
Clarksville, Indiana
St. Louis
Louisville, Kentucky
Charlottesville, Virginia
SPANISH TERRITORY
UNITED STATES
MISSISSIPPI TERRITORY
North America, circa 1804
N
W
E
S


BRITISH TERRITORY
OREGON COUNTRY
Fort Clatsop, Oregon
Lewiston, Idaho
Great Falls, Montana
New Town, North Dakota
Billings, Montana
Circle of Cultures, North Dakota
ILLINOIS COUNTRY
MICH. CNTRY
Oacoma/Chamberlain, South Dakota
LOUISIANA PURCHASE
Omaha, Nebraska
Kansas City, Missouri
St. Charles
Hartford, Illinois
St. Louis
Clarksville, Indiana
Louisville, Kentucky
Charlottesville, Virginia
SPANISH TERRITORY
UNITED STATES
MISSISSIPPI TERRITORY
North America, circa 1804
N
S
E
W

Pittsburgh, Pennsylvania

Pittsburgh, Pennsylvania

Reynolds, Girard invent System F (1970s)

Impredicative polymorphism $\forall X.\, F$

Beyond (current) ordinal analysis: powerful!


BRITISH TERRITORY
OREGON COUNTRY
Fort Clatsop, Oregon
Lewiston, Idaho
Great Falls, Montana
Billings, Montana
New Town, North Dakota
Circle of Cultures, North Dakota
Oacoma/Chamberlain, South Dakota
LOUISIANA PURCHASE
Omaha, Nebraska
Kansas City, Missouri
St. Charles
St. Louis
Hartford, Illinois
Clarksville, Indiana
Louisville, Kentucky
Charlottesville, Virginia
ILLINOIS COUNTRY
MICH. CNTRY
SPANISH TERRITORY
UNITED STATES
MISSISSIPPI TERRITORY
N
S
E
W
North America, circa 1804

St. Louis, Missouri

St. Louis, Missouri

Coquand, Huet: Calculus of Constructions (1988)

Add dependent types $\Pi x : A. B$

No induction [Geuvers 2001]


BRITISH TERRITORY
OREGON COUNTRY
Fort Clatsop, Oregon
Lewiston, Idaho
Great Falls, Montana
Billings, Montana
New Town, North Dakota
Circle of Cultures, North Dakota
ILLINOIS COUNTRY
MICH. CNTRY
Oacoma/Chamberlain, South Dakota
LOUISIANA PURCHASE
Omaha, Nebraska
Kansas City, Missouri
St. Charles
Hartford, Illinois
St. Louis
Clarksville, Indiana
Louisville, Kentucky
Charlottesville, Virginia
SPANISH TERRITORY
UNITED STATES
MISSISSIPPI TERRITORY
N
W
E
S
North America, circa 1804

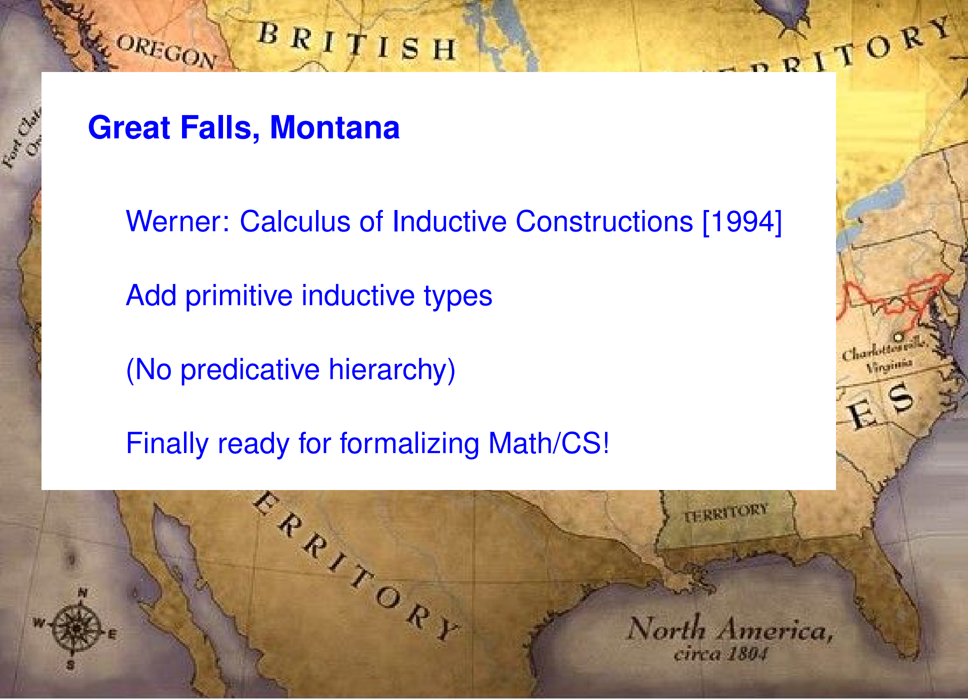
Council Bluffs, Iowa

Luo: Extended Calculus of Constructions (1990)

Add predicative hierarchy *Prop*, $Type_j$, $j \in \mathbb{N}$

Extend impredicativity $\underline{\Pi x : Type_j .\ P} : Prop$


BRITISH TERRITORY
OREGON COUNTRY
Fort Clatsop, Oregon
Lewiston, Idaho
Great Falls, Montana
Billings, Montana
New Town, North Dakota
Circle of Cultures, North Dakota
ILLINOIS COUNTRY
MICH. CNTRY
Oacoma/Chamberlain, South Dakota
LOUISIANA PURCHASE
Omaha, Nebraska
Kansas City, Missouri
St. Charles
Hartford, Illinois
St. Louis
Clarksville, Indiana
Louisville, Kentucky
Charlottesville, Virginia
SPANISH TERRITORY
UNITED STATES
MISSISSIPPI TERRITORY
North America, circa 1804
N
S
E
W



Great Falls, Montana

Werner: Calculus of Inductive Constructions [1994]

Add primitive inductive types

(No predicative hierarchy)

Finally ready for formalizing Math/CS!

Pacific Ocean: Coq

Wait a second!

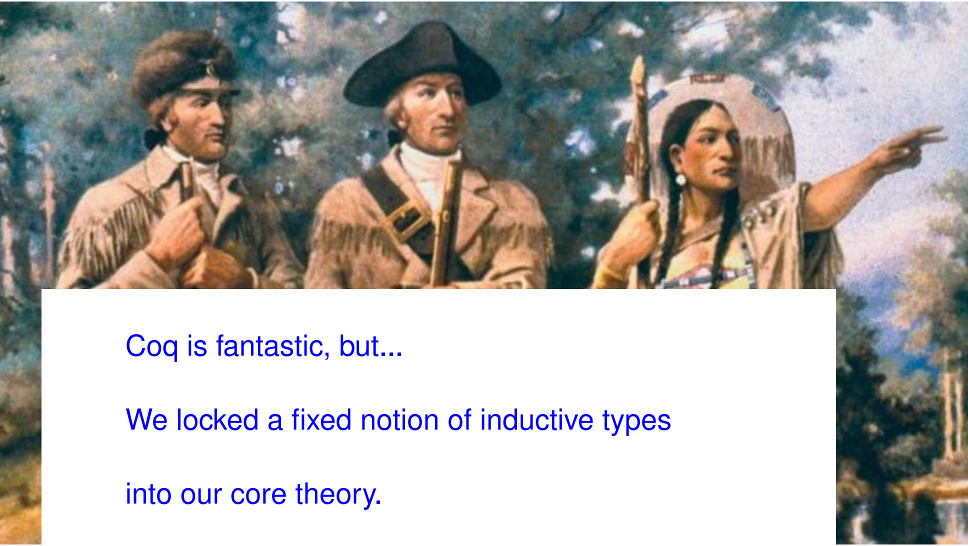
Wait a second!

Coq is fantastic, but...

We locked a fixed notion of inductive types

into our core theory.

Wait a second!



Coq is fantastic, but...

We locked a fixed notion of inductive types

into our core theory.

Should we maybe try the Platte through Nebraska?

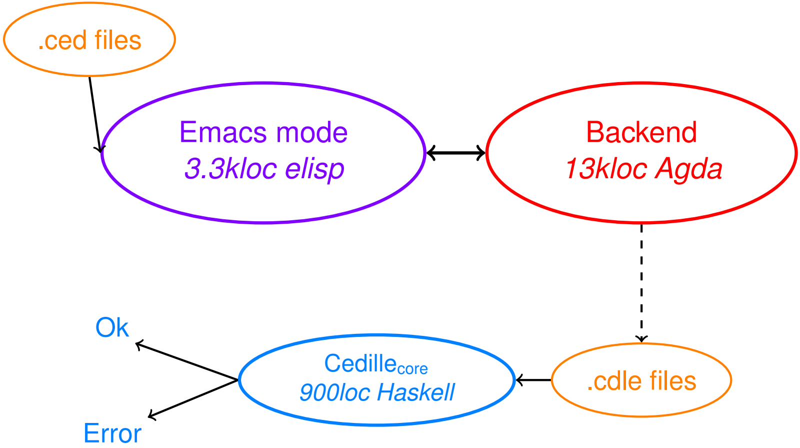
Introducing Cedille

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(!)
- ▷ Since Cedille 1.1, datatype notations, elaborated to
- ▷ Inductive, efficient lambda-encodings

Architecture of Cedille

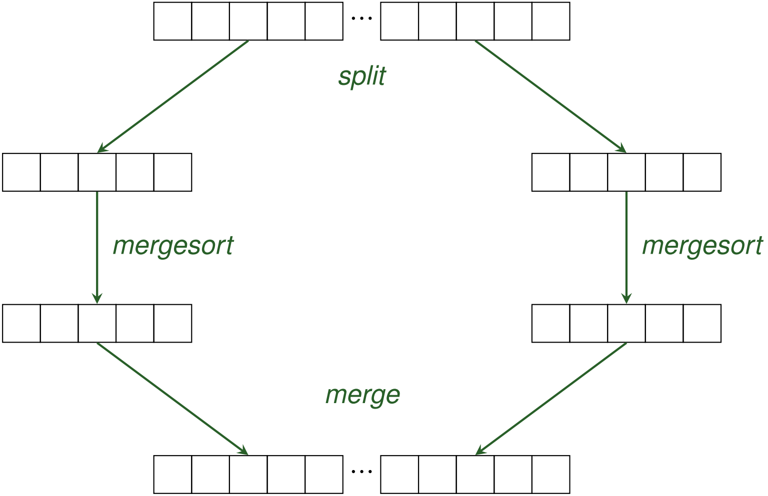


Demo

- Deriving induction
- Casts and recursive types

Histomorphic mergesort

Classic mergesort



Classic mergesort (in Haskell)

```
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = let (as,bs) = split xs
                in merge (mergeSort as) (mergeSort bs)

split (x:y:zs) = let (xs,ys) = split zs in (x:xs,y:ys)
split [x]      = ([x],[])
split []       = ([],[])
```

From `rosettacode.org`

Classic mergesort is not simple in Type Theory

```
mergeSort xs  = let (as,bs) = split xs
                in merge (mergeSort as) (mergeSort bs)
```

▷ Splitting and merging are structurally recursive.

Classic mergesort is not simple in Type Theory

```
mergeSort xs  = let (as,bs) = split xs
                in merge (mergeSort as) (mergeSort bs)
```

- ▷ Splitting and merging are structurally recursive.
- ▷ `mergeSort` itself is not.

Classic mergesort is not simple in Type Theory

```
mergeSort xs   = let (as,bs) = split xs
                 in merge (mergeSort as) (mergeSort bs)
```

- Splitting and merging are structurally recursive.
- `mergeSort` itself is not.

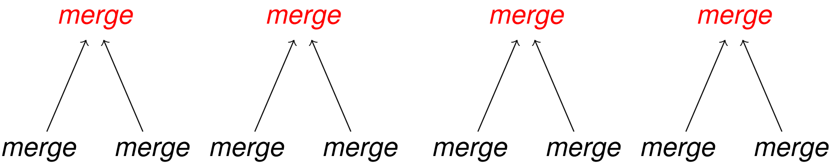
To rectify, various techniques can be applied:

- well-founded recursion
- sized types [Copello et al. 2014]
- inductive domains (cf. great survey paper “Partiality and Recursion in ITPs”, [Bove et al. 2016])

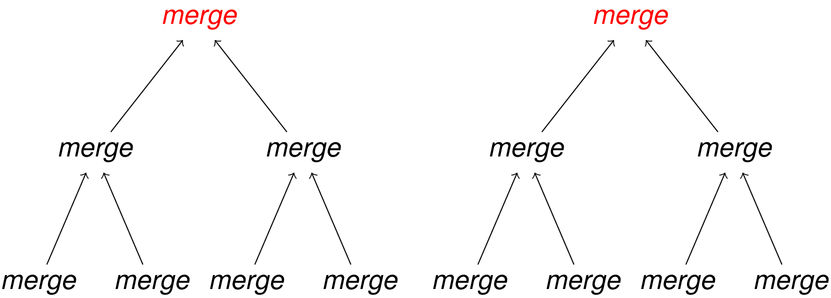
Bottom-up mergesort: a balanced tree of merges

merge *merge* *merge* *merge* *merge* *merge* *merge* *merge*

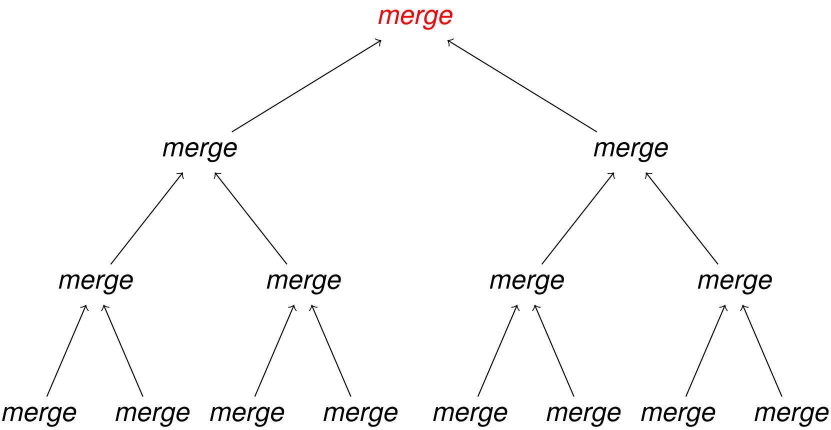
Bottom-up mergesort: a balanced tree of merges



Bottom-up mergesort: a balanced tree of merges



Bottom-up mergesort: a balanced tree of merges



Bottom-up mergesort also bad for TT

```
mergeSortBottomUp list = mergeAll (map (\x -> [x]) list)

mergeAll [sorted] = sorted
mergeAll sorteds = mergeAll (mergePairs sorteds)

mergePairs (s1 : s2 : ss) = merge s1 s2 : mergePairs ss
mergePairs sorteds = sorteds
```

From `rosettacode.org`

Bottom-up mergesort also bad for TT

```
mergeSortBottomUp list = mergeAll (map (\x -> [x]) list)

mergeAll [sorted] = sorted
mergeAll sorteds = mergeAll (mergePairs sorteds)

mergePairs (s1 : s2 : ss) = merge s1 s2 : mergePairs ss
mergePairs sorteds = sorteds
```

From `rosettacode.org`

▷ `mergeAll` not structurally recursive

Bottom-up mergesort also bad for TT

```
mergeSortBottomUp list = mergeAll (map (\x -> [x]) list)

mergeAll [sorted] = sorted
mergeAll sorteds = mergeAll (mergePairs sorteds)

mergePairs (s1 : s2 : ss) = merge s1 s2 : mergePairs ss
mergePairs sorteds = sorteds
```

From `rosettacode.org`

- `mergeAll` not structurally recursive
- though `mergePairs` decreases size of list (if non-nil)

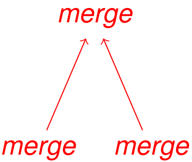
A new variant: prefix mergesort

merge

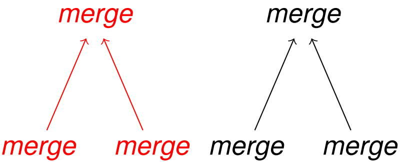
A new variant: prefix mergesort

merge *merge*

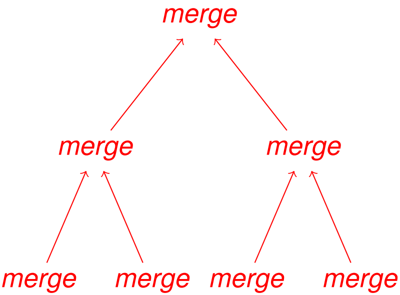
A new variant: prefix mergesort



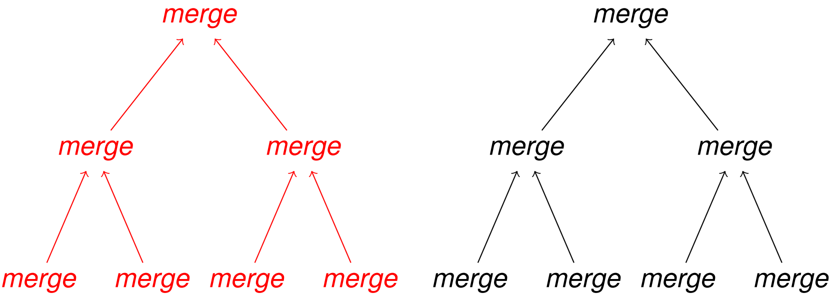
A new variant: prefix mergesort



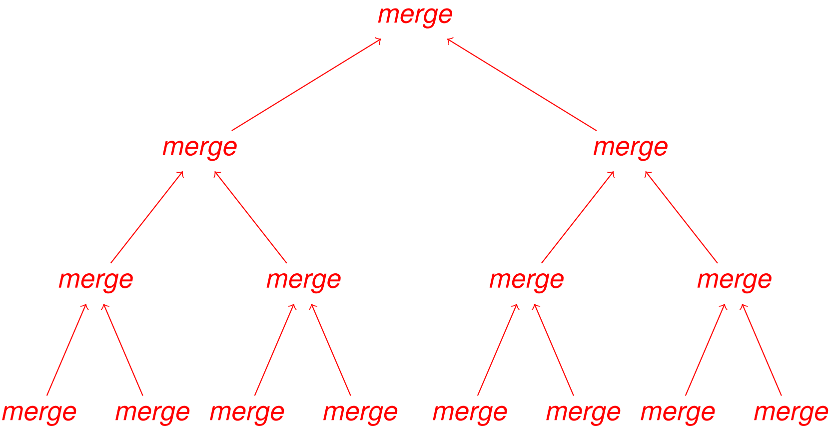
A new variant: prefix mergesort



A new variant: prefix mergesort



A new variant: prefix mergesort



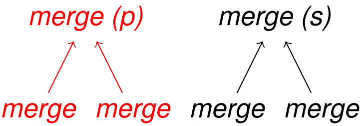
A new variant: prefix mergesort

$$\begin{tikzcd}[column sep=tiny] & \textit{merge (p)} & \\ \textit{merge} \arrow[ur] & & \textit{merge} \arrow[ul] \end{tikzcd}$$

For increasing k starting from 0:

- ▷ Let p be prefix (length 2^k) already sorted

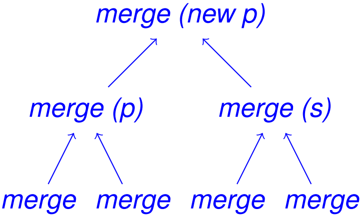
A new variant: prefix mergesort



For increasing k starting from 0:

- ▷ Let p be prefix (length 2^k) already sorted
- ▷ Let s be bottom-up mergesort of next 2^k elements

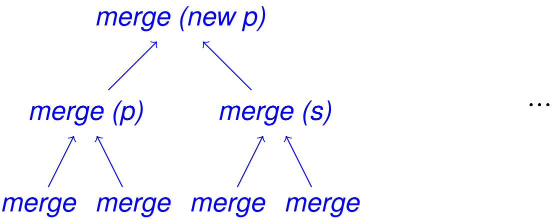
A new variant: prefix mergesort



For increasing k starting from 0:

- ▷ Let p be prefix (length 2^k) already sorted
- ▷ Let s be bottom-up mergesort of next 2^k elements
- ▷ Update p to merge of p and s

A new variant: prefix mergesort



For increasing k starting from 0:

- ▷ Let p be prefix (length 2^k) already sorted
- ▷ Let s be bottom-up mergesort of next 2^k elements
- ▷ Update p to merge of p and s
- ▷ Stop when list is empty

Prefix mergesort (in Haskell)

```
data Nat = Zero | Succ Nat

takePow2 :: Nat -> Int -> [Int] -> ([Int],[Int])
takePow2 = ...

prefixMergeSort :: [Int] -> [Int]
prefixMergeSort [] = []
prefixMergeSort (h : t) = loop t [h] Zero
  where loop [] p _ = p
        loop (h : t) p n =
          let (s,t') = takePow2 n h t in
            loop t' (merge p s) (Succ n)
```

Prefix mergesort (in Haskell)

```
data Nat = Zero | Succ Nat

takePow2 :: Nat -> Int -> [Int] -> ([Int],[Int])
takePow2 = ...

prefixMergeSort :: [Int] -> [Int]
prefixMergeSort [] = []
prefixMergeSort (h : t) = loop t [h] Zero
  where loop [] p _ = p
        loop (h : t) p n =
          let (s,t') = takePow2 n h t in
            loop t' (merge p s) (Succ n)
```

▷ `takePow2` structurally recursive on the Nat

Prefix mergesort (in Haskell)

```
data Nat = Zero | Succ Nat

takePow2 :: Nat -> Int -> [Int] -> ([Int],[Int])
takePow2 = ...

prefixMergeSort :: [Int] -> [Int]
prefixMergeSort [] = []
prefixMergeSort (h : t) = loop t [h] Zero
  where loop [] p _ = p
        loop (h : t) p n =
          let (s,t') = takePow2 n h t in
            loop t' (merge p s) (Succ n)
```

- ▷ `takePow2` structurally recursive on the Nat
- ▷ `takePow2` takes head and tail, to assure that

Prefix mergesort (in Haskell)

```
data Nat = Zero | Succ Nat

takePow2 :: Nat -> Int -> [Int] -> ([Int],[Int])
takePow2 = ...

prefixMergeSort :: [Int] -> [Int]
prefixMergeSort [] = []
prefixMergeSort (h : t) = loop t [h] Zero
  where loop [] p _ = p
        loop (h : t) p n =
          let (s,t') = takePow2 n h t in
            loop t' (merge p s) (Succ n)
```

- ▷ `takePow2` structurally recursive on the Nat
- ▷ `takePow2` takes head and tail, to assure that
- ▷ `t'` is a sublist of `t`

Prefix mergesort (in Haskell)

```
data Nat = Zero | Succ Nat

takePow2 :: Nat -> Int -> [Int] -> ([Int],[Int])
takePow2 = ...

prefixMergeSort :: [Int] -> [Int]
prefixMergeSort [] = []
prefixMergeSort (h : t) = loop t [h] Zero
  where loop [] p _ = p
        loop (h : t) p n =
          let (s,t') = takePow2 n h t in
            loop t' (merge p s) (Succ n)
```

- `takePow2` structurally recursive on the Nat
- `takePow2` takes head and tail, to assure that
- `t'` is a sublist of `t`
- So recursive call to `loop` is structurally decreasing (`h : t ≻ t'`)

How do we code this in Cedille?

- Cedille implements histomorphic recursion
- Pattern-matching recursions on inductive data *D* provide
  - An abstract type *A*
  - A function to use for recursive calls on type *A*
  - Evidence that *A* is *D*-like
    - can be decomposed like *D*, and
    - cast to *D*
- Datatype declarations add predicate for *D*-like, some helpers

The `List` datatype

```
data List (A: ⋆): ⋆ =
  | nil: List
  | cons: A → List → List.
```

This declaration introduces:

`Is/List`	:	Π `A : ⋆ . ⋆ → ⋆`	predicate for list-like

The `List` datatype

```
data List (A: ⋆): ⋆ =
  | nil: List
  | cons: A → List → List.
```

This declaration introduces:

`Is/List`	:	`Π A : ⋆ . ⋆ → ⋆`	predicate for list-like
`is/List`	:	`∀ A : ⋆ . Is/List ·A ·(List ·A)`	evidence that `List` is list-like

The `List` datatype

```
data List (A: ⋆): ⋆ =
  | nil: List
  | cons: A → List → List.
```

This declaration introduces:

`Is/List`	:	`Π A : ⋆ . ⋆ → ⋆`	predicate for list-like
`is/List`	:	`∀ A : ⋆ . Is/List ·A ·(List ·A)`	evidence that `List` is list-like
`to/List`	:	`∀ A : ⋆ . ∀ X : ⋆ . & Is/List ·A ·X ⇒ X → List ·A`	cast from list-like `X` to `List`

The `loop` helper function for prefix mergesort

Haskell:

```
loop [] p _ = p
loop (h : t) p n =
  let (s,t') = takePow2 n h t in
    loop t' (merge p s) (Succ n)
```

Cedille:

```
μ loop . t
  @ (λ _ : List · Nat . List · Nat → Nat → List · Nat)
  { nil → λ p . λ _ . p
  | cons h t → λ p . λ n .
      μ' takePow2 -isType/loop n h t
      { pair s t' → loop t' (merge p s) (succ n) }}
```

- μ introduces pattern-matching recursion
  - always over some given data, here a list `t`
- μ' is a simple pattern-match

What do you get for histomorphic recursion?

```
μ loop . t
  @ (λ _ : List · Nat . List · Nat → Nat → List · Nat)
  { nil → λ p . λ _ . p
  | cons h t → λ p . λ n .
      μ′ takePow2 -isType/loop n h t
      { pair s t′ → loop t′ (merge p s) (succ n) }}
```

The following are available in the body of `cons`-clause:

`Type/loop`	:	$\star$	abstract type for subdata

What do you get for histomorphic recursion?

```
μ loop . t
  @ (λ _ : List · Nat . List · Nat → Nat → List · Nat)
  { nil → λ p . λ _ . p
  | cons h t → λ p . λ n .
      μ′ takePow2 -isType/loop n h t
      { pair s t′ → loop t′ (merge p s) (succ n) }}
```

The following are available in the body of `cons`-clause:

`Type/loop`	:	$\star$	abstract type for subdata
`t`	:	`Type/loop`	subdata (tail of list), at abstract type

What do you get for histomorphic recursion?

```
μ loop . t
  @ (λ _ : List · Nat . List · Nat → Nat → List · Nat)
  { nil → λ p . λ _ . p
  | cons h t → λ p . λ n .
      μ′ takePow2 -isType/loop n h t
      { pair s t′ → loop t′ (merge p s) (succ n) }}
```

The following are available in the body of `cons`-clause:

`Type/loop`	:	⋆	abstract type for subdata
`t`	:	`Type/loop`	subdata (tail of list), at abstract type
`loop`	:	`Type/loop → List ·Nat → Nat → List ·Nat`	for recursive calls

What do you get for histomorphic recursion?

```
μ loop . t
  @ (λ _ : List · Nat . List · Nat → Nat → List · Nat)
  { nil → λ p . λ _ . p
  | cons h t → λ p . λ n .
      μ′ takePow2 -isType/loop n h t
      { pair s t′ → loop t′ (merge p s) (succ n) }}
```

The following are available in the body of `cons`-clause:

`Type/loop`	:	`⋆`	abstract type for subdata
`t`	:	`Type/loop`	subdata (tail of list), at abstract type
`loop`	:	`Type/loop → List ·Nat → Nat → List ·Nat`	for recursive calls
`isType/loop`	:	`Is/List ·Nat ·Type/loop`	evidence that abstract type is list-like

takePow2, using Is/List

```
takePow2 : ∀ T : ⋆ . Is/List · Nat · T ⇒
           Nat → Nat → T → Pair · (List · Nat) · T =
  Λ T . Λ mT . λ n .
    μ takePow2 . n
      { zero → λ a . λ l .  pair (singleton a) l
      | succ n → λ a . λ l .
          [p = takePow2 n a l ] -
          μ′ p
          { pair t1 l →
            μ′<mT> l
             { nil → p
             | cons a l →
               μ′ (takePow2 n a l)
                { pair t2 l →
                  pair (merge t1 t2) l }}}} .
```

takePow2, using Is/List

```
takePow2 : ∀ T : ⋆ . Is/List · Nat · T ⇒
           Nat → Nat → T → Pair · (List · Nat) · T =
  Λ T . Λ mT . λ n .
    μ takePow2 . n
      { zero → λ a . λ l .  pair (singleton a) l
      | succ n → λ a . λ l .
        [p = takePow2 n a l ] -
         μ′ p
         { pair t1 l →
           μ′<mT> l
            { nil → p
            | cons a l →
              μ′ (takePow2 n a l)
               { pair t2 l →
                 pair (merge t1 t2) l }}}} .
```

- μ' accepts evidence that T is list-like!
- So you can use pattern-matching on *D*-like types

Types for histomorphic mergesort

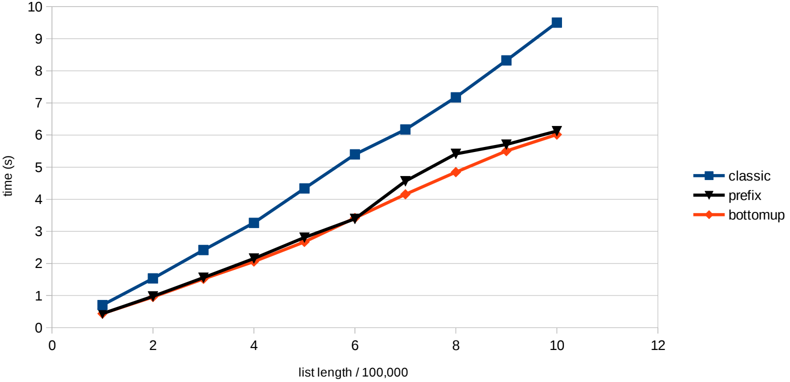
```
merge : List ·Nat → List ·Nat → List ·Nat

takePow2 : ∀ T : ⋆ . Is/List · Nat · T ⇒
           Nat → Nat → T → Pair · (List · Nat) · T

msort : List · Nat → List · Nat
  loop : List · Nat → List · Nat → Nat → List · Nat
```

- Use `Is/List` and μ' to define `takePow2` outside of `msort`
- `msort` can recurse on values of type `Type/List` returned by `takePow2`
- Nothing else needed to convince the (implicit) termination checker!
- More flexible than nested recursions/lexicographic recursions

Comparing mergesort variants in Haskell



For each list length,

- ▷ Randomly generate 10 lists of that length
- ▷ Run each sorting algorithm (as a separate process)
- ▷ List elements range from min to max `Int`

Ç Current and future directions

Schematic Cedille *(Cedille 1.2)*

When you run out of power...

Because you must at some point, if sound [Gödel 1931]

- ▷ Predicative hierarchy?
  - Complexities with level expressions
  - Temptation to keep going (universe-polymorphism!)
- ▷ A time-honored alternative:

When you run out of power...

Because you must at some point, if sound [Gödel 1931]

- ▷ Predicative hierarchy?
  - Complexities with level expressions
  - Temptation to keep going (universe-polymorphism!)
- ▷ A time-honored alternative: **go schematic!**

What does schematic mean?

Allowing parametrized definitions not expressible in the language.

In type theory: definitions exceeding the allowed products.

What does schematic mean?

Allowing parametrized definitions not expressible in the language.

In type theory: definitions exceeding the allowed products.

A simple example in current Cedille: parametrized kind definitions.

κ `(I : ⋆) = I → ⋆.`

- In Cedille, λ `I : ⋆ . I → ⋆` is not typable (no $\star \rightarrow \Box$)
- All uses of κ must include an argument for `I`
- So such definitions work like typed macros
- But we want to go beyond this...

Telescope-generic developments

- We are working on parametrization by <u>telescopes</u>
  - A telescope is a dependent sequence of declarations, e.g.

    `(A : ⋆)(a : A)`
  - Allow definitions and modules to be parametric in γ `: tel`
  - Form products over such γ, with λ-abstractions and applications
- Benefits:
  - More generic developments (avoid duplication), especially
  - Lambda-encodings with different parameter/index lists
  - Eliminate complex code in Cedille for translating datatypes,
  - Replacing with telescope-generic Cedille

Schematic `RecType`

For recursive indexed types:

```
module RecType (γ : tel) (F : (γ → ⋆) → γ → ⋆).

Cast : (γ → ⋆) → (γ → ⋆) → ⋆ = ...

Mono : ⋆ = ∀ A: γ → ⋆. ∀ B: γ → ⋆.
           Cast ·A ·B ⇒ Cast ·(F ·A) ·(F ·B).

Rec : γ → ⋆ = λ τ: γ. ∀ A: γ → ⋆. Cast ·(F ·A) ·A ⇒ A τ.
```

Instantiate γ by `(n : Nat)` for `Nat`-indexed recursive types (e.g.)

Native disjunctions and existentials *(Cedille 1.3)*

We can already encode existential types

```
module WeakSigma(A : ⋆)(B : A → ⋆).

Exists : ⋆ = ∀ X : ⋆ . (∀ a : A . B a → X) → X .
witness : ∀ a : A . B a → wSigma =
  Λ a . λ b . Λ X . λ c . c -a b .
```

- But `witness -a b` normalizes to λ `c . c b`
- For some situations, you really want just `b`
- Also, we are considering *n*-ary disjunctions
  - Introductions $\mathtt{inj}_{\mathtt{k}}\ t$
  - As opposed to $\mathtt{inj}_2\ (\cdots\ \mathtt{inj}_2\ (\mathtt{inj}_1 t))$ for binary
- Long bad history in proof theory...

The unpleasant eliminations

$$\frac{\Gamma \vdash t : \exists x : T.F \qquad \Gamma, x : T, u : F \vdash t' : C}{\Gamma \vdash \mathit{unpack}\ t\ \mathit{as}\ (x.y.t') : C}$$

$$\frac{\Gamma \vdash t : F_1 \vee F_2 \qquad \Gamma, u : F_1 \vdash t_1 : C \qquad \Gamma, v : F_2 \vdash t_2 : C}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ (u.t_1, v.t_2) : C}$$

- Host of well-known issues (commuting conversions, etc.)
- Like a cut of sequent calculus

The unpleasant eliminations

$$\frac{\Gamma \vdash t : \exists x : T.F \qquad \Gamma, x : T, u : F \vdash t' : C}{\Gamma \vdash \mathit{unpack}\ t\ \mathit{as}\ (x.y.t') : C}$$

$$\frac{\Gamma \vdash t : F_1 \vee F_2 \qquad \Gamma, u : F_1 \vdash t_1 : C \qquad \Gamma, v : F_2 \vdash t_2 : C}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ (u.t_1, v.t_2) : C}$$

- ▷ Host of well-known issues (commuting conversions, etc.)
- ▷ Like a cut of sequent calculus …

The unpleasant eliminations

$$\frac{\Gamma \vdash t : \exists x : T.F \qquad \Gamma, x : T, u : F \vdash t' : C}{\Gamma \vdash \mathit{unpack}\ t\ \mathit{as}\ (x.y.t') : C}$$

$$\frac{\Gamma \vdash t : F_1 \vee F_2 \qquad \Gamma, u : F_1 \vdash t_1 : C \qquad \Gamma, v : F_2 \vdash t_2 : C}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ (u.t_1, v.t_2) : C}$$

- ▷ Host of well-known issues (commuting conversions, etc.)
- ▷ Like a cut of sequent calculus . . . **!**

Sequent calculus and natural deduction

- ▷ Natural deduction as emerging from sequent calculus
  - (Historically the reverse.)
  - Left-rules of seq. calc. give rise (somehow!) to eliminations
  - We pivot seq. calc. proofs so left-rules go to the top
  - Eliminations followed by introductions (normal proofs)
- ▷ For good connectives, this works ($\to$, $\wedge$, $\top$, $\bot$, $\forall$)
- ▷ For $\vee$ and $\exists$, we get stuck

A Curry-style perspective

Let us assign natural-deduction terms to sequent proofs

- Right-rules get assigned introduction forms
- Left-rules get elimination forms... but where? how?

A Curry-style perspective

Let us assign natural-deduction terms to sequent proofs

Right-rules get assigned introduction forms

Left-rules get elimination forms... but where? how?

$$
\begin{array}{lll}
\textit{formulas } T & ::= & T \to T' \mid T \wedge T' \mid T \vee T' \mid \cdots \\
\textit{terms } t & ::= & \lambda x.\, t \mid t\ t' \mid t.1 \mid t.2 \mid (t, t') \mid \\
 & & t.(1) \mid t.(2) \mid t \| t' \mid \cdots
\end{array}
$$

A Curry-style perspective

Let us assign natural-deduction terms to sequent proofs

Right-rules get assigned introduction forms

Left-rules get elimination forms... but where? how?

formulas T	$::=$	$T \to T' \mid T \wedge T' \mid T \vee T' \mid \cdots$
terms t	$::=$	$\lambda x.\, t \mid t\ t' \mid t.1 \mid t.2 \mid (t, t') \mid$
		$t.(1) \mid t.(2) \mid t \| t' \mid \cdots$
contexts Γ	$::=$	$\cdot \mid \boxed{\mathrm{t : T}}$

A Curry-style perspective

Let us assign natural-deduction terms to sequent proofs

Right-rules get assigned introduction forms

Left-rules get elimination forms... but where? how?

$$
\begin{array}{lll}
\textit{formulas } T & ::= & T \to T' \mid T \wedge T' \mid T \vee T' \mid \cdots \\
\textit{terms } t & ::= & \lambda x.\, t \mid t\ t' \mid t.1 \mid t.2 \mid (t, t') \mid \\
 & & t.(1) \mid t.(2) \mid t \| t' \mid \cdots \\
\textit{contexts } \Gamma & ::= & \cdot \mid \boxed{\mathrm{t : T}}
\end{array}
$$

$$
\frac{(t : T) \in \Gamma}{\Gamma \vdash t : T}\ ax
\qquad
\frac{\Gamma, t : T_1 \wedge T_2, t.1 : T_1, t.2 : T_2 \vdash t' : C}{\Gamma, t : T_1 \wedge T_2 \vdash t' : C}\ \wedge L
$$

A Curry-style perspective

Let us assign natural-deduction terms to sequent proofs

Right-rules get assigned introduction forms

Left-rules get elimination forms... but where? how?

$$
\begin{array}{lcl}
\textit{formulas } T & ::= & T \to T' \mid T \wedge T' \mid T \vee T' \mid \cdots \\
\textit{terms } t & ::= & \lambda\, x.\, t \mid t\; t' \mid t.1 \mid t.2 \mid (t, t') \mid \\
 & & t.(1) \mid t.(2) \mid t \| t' \mid \cdots \\
\textit{contexts } \Gamma & ::= & \cdot \mid \boxed{\mathrm{t : T}}
\end{array}
$$

$$
\frac{(t : T) \in \Gamma}{\Gamma \vdash t : T}\ ax
\qquad
\frac{\Gamma, t : T_1 \wedge T_2, t.1 : T_1, t.2 : T_2 \vdash t' : C}{\Gamma, t : T_1 \wedge T_2 \vdash t' : C}\ \wedge L
$$

$$
\frac{\Gamma, t.(1) : T_1 \vdash t_1 : C \qquad \Gamma, t.(2) : T_2 \vdash t_2 : C}{\Gamma, t : T_1 \vee T_2 \vdash t_1 \| t_2 : C}\ \vee L
$$

Co-projections

$$\frac{\Gamma, t.(1) : T_1 \vdash t_1 : C \qquad \Gamma, t.(2) : T_2 \vdash t_2 : C}{\Gamma, t : T_1 \vee T_2 \vdash t_1 \| t_2 : C} \vee L$$

Eliminations $t.(1)$ and $t.(2)$, introductions $inj_1\ t$, $inj_2\ t$

The $(\vee L)$ rule ensures that one branch of $t_1 \| t_2$ will succeed

- Failure is reducing $(inj_1\ t).(2)$ or $(inj_2\ t).(1)$

Not sure yet how to formulate natural-deduction typing for these

Switch to Sequent Calculus *(Cedille 2.0)*

Cuts and control!

- ▷ Cuts give rise to control operators classically
- ▷ Good reasons for avoid classicality for **programs**
  - Under Curry-Howard, $T \vee T'$ is a sum type
  - But classically, get no information splitting on $T \vee \neg T$
- ▷ Can we support control and retain canonicity?
- ▷ Can we also achieve perfect duality?
  - Logics like BiInt have duality and control, but
  - Not canonicity!
  - Can prove $F \vee (\top \prec F)$
- ▷ Modifying a system of Wansing's, we have a candidate...

Cuts and control!

- Cuts give rise to control operators classically
- Good reasons for avoid classicality for **programs**
  - Under Curry-Howard, $T \vee T'$ is a sum type
  - But classically, get no information splitting on $T \vee \neg T$
- Can we support control and retain canonicity?
- Can we also achieve perfect duality?
  - Logics like BiInt have duality and control, but
  - Not canonicity!
  - Can prove $F \vee (\top \prec F)$
- Modifying a system of Wansing's, we have a candidate... to be discussed another time!

Conclusion: our route

Following the Missouri River (*CC*),

We took the North Platte from Omaha (Curry-style $CC + \iota, \simeq, \forall$)

Heading south (hotter? histomorphic recursion)

to the Colorado River (getting crazy)

towards the Pacific (ultimate Type Theory)

Right now we are maybe passing through...

Acknowledgments

- Current Cedille dev team:
  - Postdoc: Stephan Spahn
  - Doctoral: Andrew Marmaduke, Chris Jenkins, Tony Cantor
  - High schooler: Colin McDonald
- Group alums: Denis Firsov, Larry Diehl, Ernesto Copello, Richard Blair, Ananda Guneratne, Chad Reynolds, Matthew Heimerdinger
- Funders: NSF 1524519, DoD FA9550-16-1-0082

AMDG