Primitively (co)recursive definitions for Isabelle/HOL

Lorenz Panny Jasmin C. Blanchette Dmitriy Traytel

Fakultät für Informatik
Technische Universität München

Datatypes

datatype_new α *list* $=$ Nil $\mid$ Cons α (α *list*)

Datatypes

datatype_new α *list* $=$ Nil $\mid$ Cons α (α *list*)

datatype_new α *list* $=$ null: Nil $\mid$ Cons (hd: α) (tl: α *list*)

Codatatypes

- Datatype values: finite
- Codatatype values: possibly infinite

Codatatypes

- Datatype values: finite
- Codatatype values: possibly infinite

codatatype α *llist* = lnull : LNil | LCons (lhd : α) (ltl : α *llist*)

Codatatypes

- Datatype values: finite
- Codatatype values: possibly infinite

codatatype α *llist* $=$ lnull : LNil $\mid$ LCons (lhd : α) (ltl : α *llist*)

codatatype α *stream* $=$ SCons (shd : α) (stl : α *stream*)

(Co)datatypes: Mutual (co)recursion

datatype_new $\alpha\ tree_M$ = TEmpty_M | $\mathsf{TNode}_M\ \alpha\ (\alpha\ forest)$
and $\alpha\ forest$ = FNil | FCons $(\alpha\ tree_M)\ (\alpha\ forest)$

codatatype $\alpha\ ltree_M$ = $\mathsf{LTEmpty}_M$ | $\mathsf{LTNode}_M\ \alpha\ (\alpha\ lforest)$
and $\alpha\ lforest$ = LFNil | LFCons $(\alpha\ ltree_M)\ (\alpha\ lforest)$

(Co)datatypes: Nested (co)recursion

datatype_new α *tree* = TEmpty | TNode (α *tree list*)

codatatype α *ltree* = LTEmpty | LTNode (α *ltree llist*)

Primitive recursion: **primrec**

primrec

- automates tedious recursor-based definition

Primitive recursion: **primrec**

primrec

- automates tedious recursor-based definition
- uses recursor theorems to prove characteristic theorems

Primitive recursion: **primrec**

primrec

- automates tedious recursor-based definition
- uses recursor theorems to prove characteristic theorems
- supports nested recursion through map functions (new)

Primitive recursion: **primrec**

primrec

- automates tedious recursor-based definition
- uses recursor theorems to prove characteristic theorems
- supports nested recursion through map functions (new)
  ...in addition to nested-as-mutual recursion

Primitive recursion

Primitively recursive functions

- consume a datatype value one constructor in each recursive descent

Primitive recursion

Primitively recursive functions

- consume a datatype value one constructor in each recursive descent
- use pattern matching syntax f (C x) $=$ *rhs* to ensure C's consumption

Primitive recursion

Primitively recursive functions

- consume a datatype value one constructor in each recursive descent
- use pattern matching syntax $\mathsf{f}\ (\mathsf{C}\ x) = \mathit{rhs}$ to ensure C's consumption
- can be recursively applied to unmodified constructor arguments
  - Good: $\mathsf{f}\ (\mathsf{C}\ x) = g\ (\mathsf{f}\ x)$
  - Bad: $\mathsf{f}\ (\mathsf{C}\ x) = \mathsf{f}\ (g\ x)$
- also via a map function (nested recursion)
  - Good: $\mathsf{f}\ (\mathsf{C}\ xs) = \mathsf{map}\ (g \circ \mathsf{f})\ xs$
  - Bad: $\mathsf{f}\ (\mathsf{C}\ xs) = \mathsf{map}\ (\mathsf{f} \circ g)\ xs$

Primitive recursion: Examples

```
primrec list_sum :: nat list ⇒ nat where
  list_sum Nil = 0
| list_sum (Cons x xs) = x + list_sum xs
```

Primitive recursion: Examples

```
primrec list_sum :: nat list ⇒ nat where
  list_sum Nil = 0
| list_sum (Cons x xs) = x + list_sum xs
```

```
primrec tree_M_sum :: nat tree_M ⇒ nat
    and forest_sum :: nat forest ⇒ nat where
  tree_M_sum TEmpty_M = 0
| tree_M_sum (TNode_M v cs) = v + forest_sum cs
| forest_sum FNil = 0
| forest_sum (FCons t ts) = tree_M_sum t + forest_sum ts
```

Primitive recursion: Examples (nested)

```
primrec list_sum :: nat list ⇒ nat where
  list_sum Nil = 0
| list_sum (Cons x xs) = x + list_sum xs
```

```
primrec tree_sum :: nat tree ⇒ nat where
  tree_sum TEmpty = 0
| tree_sum (TNode v cs) = v + list_sum (map tree_sum cs)
```

Primitive recursion: Recursors

$$\text{rec\_list} :: \alpha \Rightarrow (\beta \Rightarrow \beta\ list \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \beta\ list \Rightarrow \alpha$$

$$\text{rec\_list}\ n\ c\ \text{Nil} = n$$
$$\text{rec\_list}\ n\ c\ (\text{Cons}\ x\ xs) = c\ x\ xs\ (\text{rec\_list}\ n\ c\ xs)$$

Primitive recursion: Recursors

$\mathsf{rec\_list} :: \alpha \Rightarrow (\beta \Rightarrow \beta\ \mathit{list} \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \beta\ \mathit{list} \Rightarrow \alpha$

$\mathsf{rec\_list}\ n\ c\ \mathsf{Nil} = n$
$\mathsf{rec\_list}\ n\ c\ (\mathsf{Cons}\ x\ xs) = c\ x\ xs\ (\mathsf{rec\_list}\ n\ c\ xs)$

definition length $:: \alpha\ \mathit{list} \Rightarrow \mathit{nat}$ **where**
$\mathsf{length} \equiv \mathsf{rec\_list}\ 0\ (\lambda x\ xs\ r.\ 1 + r)$

Primitive recursion: Recursors

```
rec_list :: α ⇒ (β ⇒ β list ⇒ α ⇒ α) ⇒ β list ⇒ α

rec_list n c Nil = n
rec_list n c (Cons x xs) = c x xs (rec_list n c xs)
```

```
definition length :: α list ⇒ nat where
length ≡ rec_list 0 (λx xs r. 1 + r)
```

```
lemma length Nil = 0
  unfolding length_def list.rec(1) ..

lemma length (Cons x xs) = 1 + length xs
  unfolding length_def list.rec(2) ..
```

Primitive recursion: Recursors (nested)

```
rec_tree :: α ⇒ (β ⇒ (β tree × α) list ⇒ α) ⇒ β tree ⇒ α

rec_tree e n TEmpty = e
rec_tree e n (TNode v cs) = n v (map (λt. (t, rec_tree e n t)) cs)
```

```
definition tree_sum :: nat tree ⇒ nat where
tree_sum ≡ rec_tree 0 (λv cs. v + list_sum (map snd cs))
```

```
lemma tree_sum TEmpty = 0
  unfolding tree_sum_def tree.rec(1) ..

lemma tree_sum (TNode v cs) = v + list_sum (map tree_sum cs)
  unfolding tree_sum_def tree.rec(2) list.map_comp comp_def snd_conv ..
```

Primitive corecursion

Primitively corecursive functions...

- produce a codatatype value one constructor in each corecursive descent

Primitive corecursion

Primitively corecursive functions...

- produce a codatatype value one constructor in each corecursive descent
- use syntactic restrictions to ensure emission of constructors

Primitive corecursion

Primitively corecursive functions...

- produce a codatatype value one constructor in each corecursive descent
- use syntactic restrictions to ensure emission of constructors
- can be corecursively applied to any arguments, but must be outermost call in a constructor argument
  - Bad: $\mathsf{f}\ \mathit{args} = \mathsf{f}\ \mathit{args}'$
  - Good: $\mathsf{f}\ \mathit{args} = \mathsf{C}\ (\mathsf{f}\ \mathit{args}')$
  - Bad: $\mathsf{f}\ \mathit{args} = \mathsf{C}\ (g\ (\mathsf{f}\ \mathit{args}'))$

Primitive corecursion

Primitively corecursive functions...

- produce a codatatype value one constructor in each corecursive descent
- use syntactic restrictions to ensure emission of constructors
- can be corecursively applied to any arguments, but must be outermost call in a constructor argument
  - Bad: $\mathsf{f}\ \mathit{args} = \mathsf{f}\ \mathit{args}'$
  - Good: $\mathsf{f}\ \mathit{args} = \mathsf{C}\ (\mathsf{f}\ \mathit{args}')$
  - Bad: $\mathsf{f}\ \mathit{args} = \mathsf{C}\ (g\ (\mathsf{f}\ \mathit{args}'))$
  - Good: $\mathsf{f}\ \mathit{args} = \mathsf{C}\ (\mathsf{map}\ (\mathsf{f} \circ g)\ \mathit{args}')$
  - Bad: $\mathsf{f}\ \mathit{args} = \mathsf{C}\ (\mathsf{map}\ (g \circ \mathsf{f})\ \mathit{args}')$

Primitive corecursion: **primcorec**

primcorec

- automates tedious corecursor-based definition
- uses corecursor theorems to prove characteristic theorems

Primitive corecursion: Syntaxes

$\mathsf{f} :: \mathit{nat} \Rightarrow \mathit{nat\ llist}$

Primitive corecursion: Syntaxes

$\mathsf{f} :: \mathit{nat} \Rightarrow \mathit{nat\ llist}$

- Constructor view

$n = 0 \Longrightarrow \mathsf{f}\ n = \mathsf{LNil}$
$\_ \Longrightarrow \mathsf{f}\ n = \mathsf{LCons}\ n\ (\mathsf{f}\ (1 + n))$

Primitive corecursion: Syntaxes

$$\mathsf{f} :: \mathit{nat} \Rightarrow \mathit{nat\ llist}$$

- Constructor view

$$\begin{array}{l} n = 0 \Longrightarrow \mathsf{f}\ n = \mathsf{LNil} \\ \_ \Longrightarrow \mathsf{f}\ n = \mathsf{LCons}\ n\ (\mathsf{f}\ (1 + n)) \end{array}$$

- Destructor view

$$\begin{array}{l} n = 0 \Longrightarrow \mathsf{lnull}\ (\mathsf{f}\ n) \\ \mathsf{lhd}\ (\mathsf{f}\ n) = n \\ \mathsf{ltl}\ (\mathsf{f}\ n) = \mathsf{f}\ (1 + n) \end{array}$$

Primitive corecursion: Syntaxes

$$\mathsf{f} :: \mathit{nat} \Rightarrow \mathit{nat\ llist}$$

- Constructor view

$$n = 0 \Longrightarrow \mathsf{f}\ n = \mathsf{LNil}$$
$$\_ \Longrightarrow \mathsf{f}\ n = \mathsf{LCons}\ n\ (\mathsf{f}\ (1 + n))$$

- Destructor view

$$n = 0 \Longrightarrow \mathsf{lnull}\ (\mathsf{f}\ n)$$
$$\mathsf{lhd}\ (\mathsf{f}\ n) = n$$
$$\mathsf{ltl}\ (\mathsf{f}\ n) = \mathsf{f}\ (1 + n)$$

- Code view

$$\mathsf{f}\ n = (\mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{LNil}\ \mathsf{else}\ \mathsf{LCons}\ n\ (\mathsf{f}\ (1 + n)))$$

Primitive corecursion: Examples

primcorec zeroes :: *nat llist* **where**
  ¬ lnull zeroes
| lhd zeroes = 0
| ltl zeroes = zeroes

Primitive corecursion: Examples

primcorec zeroes :: *nat llist* **where**
  ¬ lnull zeroes
| lhd zeroes = 0
| ltl zeroes = zeroes

primcorec lappend :: α *llist* $\Rightarrow$ α *llist* $\Rightarrow$ α *llist* **where**
  lnull *xs* $\wedge$ lnull *ys* $\Longrightarrow$ lnull (lappend *xs ys*)
| lhd (lappend *xs ys*) = lhd (if lnull *xs* then *ys* else *xs*)
| ltl (lappend *xs ys*) = (if lnull *xs* then ltl *ys* else lappend (ltl *xs*) *ys*)

Primitive corecursion: Examples

```
primcorec zeroes :: nat llist where
  ¬ lnull zeroes
| lhd zeroes = 0
| ltl zeroes = zeroes
```

```
primcorec lappend :: α llist ⇒ α llist ⇒ α llist where
  lnull xs ∧ lnull ys ⟹ lnull (lappend xs ys)
| lhd (lappend xs ys) = lhd (if lnull xs then ys else xs)
| ltl (lappend xs ys) = (if lnull xs then ltl ys else lappend (ltl xs) ys)
```

```
primcorec iterate :: (α ⇒ α) ⇒ α ⇒ α stream where
  iterate f x = SCons x (iterate f (f x))
```

```
primcorec iterate_while :: (α ⇒ α option) ⇒ α ⇒ α llist where
  is_none (f x) ⟹ iterate_while f x = LNil
| _ ⟹ iterate_while f x = LCons x (iterate_while f (the (f x)))
```

Primitive corecursion: Examples (nested)

```
primcorec iterate_tree :: α ⇒ (α ⇒ α) ltree ⇒ α ltree where
  iterate_tree x lt =
      (case lt of LTEmpty ⇒ LTEmpty
                | LTNode v cs ⇒ LTNode (v x)
                    (map_llist (iterate_tree x ∘ map_ltree (λf. f ∘ v)) cs))
```

Primitive corecursion: Syntax reductions

destructor $\longleftarrow$ constructor $\longleftarrow$ code

Primitive corecursion: Constructor view → destructor view

$$p\ args \implies \mathsf{f}\ args\ =\ \mathsf{C}\ a_1\ \ldots\ a_n$$

Primitive corecursion: Constructor view → destructor view

$$p\ args \implies \mathsf{f}\ args = \mathsf{C}\ a_1\ \ldots\ a_n$$

$$\begin{aligned} &p\ args \implies \mathsf{is\_C}\ (\mathsf{f}\ args)\\ &\mathsf{un\_C}_1\ (\mathsf{f}\ args) = a_1\\ &\vdots\\ &\mathsf{un\_C}_n\ (\mathsf{f}\ args) = a_n \end{aligned}$$

Primitive corecursion: Code view → constructor view

$$
\begin{aligned}
&\mathsf{lappend} :: \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist}\\
&\mathsf{lappend}\ \mathit{xs}\ \mathit{ys} =\\
&\quad (\mathsf{case}\ \mathit{xs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathit{ys}\\
&\qquad\qquad \mid \mathsf{LCons}\ x\ \mathit{xs}' \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lappend}\ \mathit{xs}'\ \mathit{ys}))
\end{aligned}
$$

Primitive corecursion: Code view → constructor view

```
lappend :: α llist ⇒ α llist ⇒ α llist
lappend xs ys =
    (case xs of LNil ⇒ ys
              | LCons x xs′ ⇒ LCons x (lappend xs′ ys))
```

```
lappend :: α llist ⇒ α llist ⇒ α llist
lappend xs ys =
    (case xs of LNil ⇒ (case ys of LNil ⇒ LNil | LCons y ys′ ⇒ LCons y ys′)
              | LCons x xs′ ⇒ LCons x (lappend xs′ ys))
```

Primitive corecursion: Code view → constructor view

$$
\begin{array}{l}
\mathsf{lappend} :: \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist} \\
\mathsf{lappend}\ \mathit{xs}\ \mathit{ys} = \\
\quad (\mathsf{case}\ \mathit{xs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathit{ys} \\
\qquad\qquad \mid \mathsf{LCons}\ x\ \mathit{xs}' \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lappend}\ \mathit{xs}'\ \mathit{ys}))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lappend} :: \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist} \\
\mathsf{lappend}\ \mathit{xs}\ \mathit{ys} = \\
\quad (\mathsf{case}\ \mathit{xs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow (\mathsf{case}\ \mathit{ys}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathsf{LNil} \mid \mathsf{LCons}\ y\ \mathit{ys}' \Rightarrow \mathsf{LCons}\ y\ \mathit{ys}') \\
\qquad\qquad \mid \mathsf{LCons}\ x\ \mathit{xs}' \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lappend}\ \mathit{xs}'\ \mathit{ys}))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lnull}\ \mathit{xs} \wedge \mathsf{lnull}\ \mathit{ys} \Longrightarrow \mathsf{lappend}\ \mathit{xs}\ \mathit{ys} = \mathsf{LNil} \\
\neg\ \mathsf{lnull}\ \mathit{xs} \vee \neg\ \mathsf{lnull}\ \mathit{ys} \Longrightarrow \mathsf{lappend}\ \mathit{xs}\ \mathit{ys} = \\
\qquad \mathsf{LCons}\ (\mathsf{if}\ \mathsf{lnull}\ \mathit{xs}\ \mathsf{then}\ \mathsf{lhd}\ \mathit{ys}\ \mathsf{else}\ \mathsf{lhd}\ \mathit{xs}) \\
\qquad\qquad\qquad\qquad (\mathsf{if}\ \mathsf{lnull}\ \mathit{xs}\ \mathsf{then}\ \mathsf{ltl}\ \mathit{ys}\ \mathsf{else}\ \mathsf{lappend}\ (\mathsf{ltl}\ \mathit{xs})\ \mathit{ys})
\end{array}
$$

Primitive corecursion: Exclusiveness

- Premises to discriminator formulas must be mutually exclusive

Primitive corecursion: Exclusiveness

- Premises to discriminator formulas must be mutually exclusive
- **primcorec** tries to solve proof obligations automatically:

  primcorec ... = **primcorecursive** ... **by** auto**?**

Primitive corecursion: Exclusiveness

- Premises to discriminator formulas must be mutually exclusive
- **primcorec** tries to solve proof obligations automatically:

  primcorec ... = **primcorecursive** ... **by** auto**?**

- *sequential* option: Interpret premises in an "if–then–else if" fashion

Primitive corecursion: Exclusiveness

- Premises to discriminator formulas must be mutually exclusive
- **primcorec** tries to solve proof obligations automatically:

  primcorec ... = **primcorecursive** ... **by** auto**?**

- *sequential* option: Interpret premises in an "if–then–else if" fashion

- *exhaustive* option: Premises are a partition of True

Primitive corecursion: Corecursors

corec_llist ::
$(\beta \Rightarrow bool) \Rightarrow$
$(\beta \Rightarrow \alpha) \Rightarrow$
$(\beta \Rightarrow bool) \Rightarrow (\beta \Rightarrow \alpha\ llist) \Rightarrow (\beta \Rightarrow \beta) \Rightarrow$
$\beta \Rightarrow \alpha\ llist$

$n\ x \Longrightarrow$ corec_llist $n\ h\ s\ e\ c\ x\ =$ LNil
$\neg\ n\ x \Longrightarrow$ corec_llist $n\ h\ s\ e\ c\ x\ =$
LCons $(h\ x)$ (if $s\ x$ then $e\ x$ else corec_llist $n\ h\ s\ e\ c\ (c\ x)$)

Primitive corecursion: Corecursors (nested)

```
corec_ltree ::
  (β ⇒ bool) ⇒
    (β ⇒ α) ⇒
    (β ⇒ (α ltree + β) llist) ⇒
  β ⇒ α ltree

e x ⟹ corec_ltree e v n x = TEmpty
¬ e x ⟹ corec_ltree e v n x =
          LTNode (v x) (lmap (λr. case r of Inl t ⇒ t
                                          | Inr x' ⇒ corec_ltree e v n x') (n x))
```

Cool examples from early adopters

codatatype α *stream_monad* = Terminated (result : α)
| Read (cont : *bool* $\Rightarrow$ α *stream_monad*)

primcorec sbind ::
α *stream_monad* $\Rightarrow$ (α $\Rightarrow$ β *stream_monad*) $\Rightarrow$ β *stream_monad* **where**
sbind f g =
(if is_Terminated f then g (result f) else Read (λb. sbind (cont f b) g))

(Johannes Hölzl)

Cool examples from early adopters

codatatype α *language* $=$ Lang ($\mathfrak{o}$: *bool*) ($\mathfrak{d}$: $\alpha \Rightarrow \alpha$ *language*)

primcorec TimesLR :: α *language* $\Rightarrow$ α *language* $\Rightarrow$ ($\alpha \times$ *bool*) *language* **where**
  $\mathfrak{o}$ (TimesLR r s) $=$ ($\mathfrak{o}$ $r \wedge \mathfrak{o}$ s)
  $\mathfrak{d}$ (TimesLR r s) $=$ ($\lambda(r, s)$. TimesLR r s) $\circ$
    ($\lambda(a, b)$. (if b then ($\mathfrak{d}$ r a, s) else if $\mathfrak{o}$ r then ($\mathfrak{d}$ s a, One) else (Zero, One)))

primcorec Times_Plus :: ($\alpha \times$ *bool*) *language* $\Rightarrow$ α *language* **where**
  $\mathfrak{o}$ (Times_Plus r) $=$ $\mathfrak{o}$ r
  $\mathfrak{d}$ (Times_Plus r) $=$ (λa. Times_Plus (Plus ($\mathfrak{d}$ r (a, True)) ($\mathfrak{d}$ r (a, False))))

(Dmitriy Traytel)

Things to take home

Things to take home

- new **primrec**
  - backwards compatible with old **primrec**
  - nested recursion through map functions (new)

Things to take home

- new **primrec**
  - backwards compatible with old **primrec**
  - nested recursion through map functions (new)
- **primcorec** (new)
  - intuitive, flexible {input, output} syntax
  - {mutual, nested} corecursion

Things to take home

- new **primrec**
  - backwards compatible with old **primrec**
  - nested recursion through map functions (new)
- **primcorec** (new)
  - intuitive, flexible {input, output} syntax
  - {mutual, nested} corecursion
- part of Isabelle2014-RC0

Primitively (co)recursive definitions for Isabelle/HOL

Lorenz Panny Jasmin C. Blanchette Dmitriy Traytel

Fakultät für Informatik
Technische Universität München