10 Years of Partiality and General Recursion in Type Theory

Ana Bove

Chalmers University of Technology

DTP'10 – July 9th 2010

Claims and Disclaims

I know that I know nothing

Socrates

Claims and Disclaims

I know that I know nothing

Socrates

Thanks to Andreas Abel, Yves Bertot, Alexander Krauss, Guilhem Moulin, Milad Niqui, Matthieu Sozeau, ...

Partiality and General Recursion in Type Theory

For decidability and consistency reasons, type theory is a theory of total functions.

Mainly (total) structural recursive functions are allowed.

No immediate way of formalising partial or general recursion functions.

Partiality and General Recursion in Type Theory

For decidability and consistency reasons, type theory is a theory of total functions.

Mainly (total) structural recursive functions are allowed.

No immediate way of formalising partial or general recursion functions.

How can one formalise (and prove correct) partial and general recursion functions in a natural way in type theory?

Partial Functions

Functions not "defined" on a certain argument are not that problematic.

Partial Functions

Functions not "defined" on a certain argument are not that problematic.

Some common solutions:

Un-interesting result:

```
tail : {A : Set} → List A → List A
tail [] = []
tail (x :: xs) = xs
```

Partial Functions

Functions not "defined" on a certain argument are not that problematic.

Some common solutions:

Un-interesting result:

```
tail : {A : Set} → List A → List A
tail [] = []
tail (x :: xs) = xs
```

What would we return for `head`?

Partial Functions

Functions not "defined" on a certain argument are not that problematic.

Some common solutions:

Un-interesting result:

```
tail : {A : Set} → List A → List A
tail [] = []
tail (x :: xs) = xs
```

What would we return for `head`?

Maybe result:

```
tail : {A : Set} → List A → Maybe (List A)
tail [] = nothing
tail (x :: xs) = just xs
```

Partial Functions (Cont.)

Restricted domain:

```
data NonEmpty {A : Set} : List A → Set where
  _::_ : (x : A) (xs : List A) → NonEmpty (x :: xs)
```

```
tail : {A : Set} → {xs : List A} → NonEmpty xs → List A
tail (y :: ys) = ys
```

(Some of the methods we will see later produce similar results on this case.)

I Will not Talk About . . .

I Will not Talk About . . .

- Functions not “defined” on a certain argument

I Will not Talk About ...

- Functions not "defined" on a certain argument

- Recursion on co-inductive functions

  See for example Bertot's and Komendantskaya's work

I Will not Talk About . . .

- Functions not "defined" on a certain argument

- Recursion on co-inductive functions

  See for example Bertot's and Komendantskaya's work

- Solutions using co-inductive types

  For example Capretta's work:

```
-- The partiality monad.
data _⊥ (A : Set) : Set where
  now   : (x : A) → A ⊥
  later : (x : ∞ (A ⊥)) → A ⊥
```

I Will Talk About . . .

Some methods to deal with *(non-structural) recursive functions* in

- Agda and Coq (based on constructive type theory)
- Isabelle (based on higher-order classical logic)

I Will Talk About ...

Some methods to deal with *(non-structural) recursive functions* in

- Agda and Coq (based on constructive type theory)
- Isabelle (based on higher-order classical logic)

Two kind of methods:

- Using the existing type system
- Modifying the existing type system (if time allows)

Recursion Must Terminate!

To guarantee termination, we require each recursive call to be performed on a *smaller* argument.

For inductive data, *structure* is the standard measure used in the systems.

Otherwise we need to give the *measure* explicitly and show it is *well-founded*.

Well-Founded Recursion via Acc

Given a set A and a (well-founded) binary relation $<$ over A:

$$\frac{a : A \qquad (x : A) \to x < a \to \mathsf{Acc}(A, <, x)}{\mathsf{Acc}(A, <, a)}$$

$$\frac{\mathsf{Acc}(A, <, a) \qquad (x : A) \to \mathsf{Acc}(A, <, x) \to ((y : A) \to y < x \to P(y)) \to P(x)}{P(a)}$$

Well-Founded Recursion via Acc

Given a set A and a (well-founded) binary relation $<$ over A:

$$\frac{a : A \qquad (x : A) \to x < a \to \mathsf{Acc}(A, <, x)}{\mathsf{Acc}(A, <, a)}$$

$$\frac{\mathsf{Acc}(A, <, a) \qquad (x : A) \to \mathsf{Acc}(A, <, x) \to ((y : A) \to y < x \to P(y)) \to P(x)}{P(a)}$$

Known problems:

- Structure of the algorithm is often not the natural one
- Logical information is mixed with the computational one
- Often results in long and complicated programs (and proofs)

Smarter Termination Checkers

```
ack : ℕ → ℕ → ℕ
ack 0 m = suc m
ack (suc n) 0 = ack n 1
ack (suc n) (suc m) = ack n (ack (suc n) m)
```

```
merge : List ℕ → List ℕ → List ℕ
merge [] ys = ys
merge xs [] = xs
merge (x :: xs) (y :: ys) = if (x < y)
                            then (x :: merge xs (y :: ys))
                            else (y :: merge (x :: xs) ys)
```

Smarter Termination Checkers (Cont.)

```
f : {A : Set} → List A → List A → List A
f [] ys = []
f (x :: xs) ys = f ys xs
```

```
g : {A : Set} → List A → List A
g [] = []
g (x :: []) = []
g (x :: y :: xs) = g (x :: xs)
```

Domain Predicates (Bove/Capretta) in Agda

We define a predicate that characterises the domain of the function...

... and the function by structural rec. on the (proof of the) domain predicate.

Domain Predicates (Bove/Capretta) in Agda

We define a predicate that characterises the domain of the function...
... and the function by structural rec. on the (proof of the) domain predicate.

```
data dom : List ℕ → Set where
  dom-[] : dom []
  dom-:: : ∀ {x} {xs} →
              dom (filter (λ y → y < x) xs) →
              dom (filter (λ y → not (y < x)) xs) →
              dom (x :: xs)
```

Domain Predicates (Bove/Capretta) in Agda

We define a predicate that characterises the domain of the function...
... and the function by structural rec. on the (proof of the) domain predicate.

```
data dom : List ℕ → Set where
  dom-[] : dom []
  dom-:: : ∀ {x} {xs} →
                dom (filter (λ y → y < x) xs) →
                dom (filter (λ y → not (y < x)) xs) →
                dom (x :: xs)


quicksort : ∀ xs → dom xs → List ℕ
quicksort [] dom-[] = []
quicksort (x :: xs) (dom-:: p q) =
                quicksort (filter (λ y → y < x) xs) p ++
                x :: quicksort (filter (λ y → not (y < x)) xs) q
```

Domain Predicates and Partiality

For total functions we can “get rid” of the domain predicate:

```
all-dom : ∀ xs → dom xs

Quicksort : List ℕ → List ℕ
Quicksort xs = quicksort xs (all-dom xs)
```

Domain Predicates and Partiality

For total functions we can "get rid" of the domain predicate:

```
all-dom : ∀ xs → dom xs

Quicksort : List ℕ → List ℕ
Quicksort xs = quicksort xs (all-dom xs)
```

But we can still talk about partial functions:

```
data dom-f : ℕ → Set where
  dom-f-1 : dom-f 1
  dom-f-s : ∀ {n} → dom-f (suc (suc n)) → dom-f (suc (suc n))

f : ∀ n → dom-f n → ℕ
f .1 dom-f-1 = 0
f (suc (suc n)) (dom-f-s p) = f (suc (suc n)) p
```

Domain Predicates and Nested Recursion

Using the schema for induction-recursion definitions (Dybjer) we can define nested recursive functions. Consider McCarthy `f91` function:

```
mutual
  data dom91 : ℕ → Set where
    dom100< : ∀ {n} → 100 < n → dom91 n
    dom≤100 : ∀ {n} → n ≤ 100 → (p : dom91 (n + 11)) ->
                        dom91 (f91 (n + 11) p) → dom91 n

  f91 : ∀ n → dom91 n → ℕ
  f91 n (dom100< h) = n - 10
  f91 n (dom≤100 h p q) = f91 (f91 (n + 11) p) q
```

Domain Predicates and Proofs

The domain predicate gives us the right induction principle!

It follows the definition of the function.

Domain Predicates and Proofs

The domain predicate gives us the right induction principle!
It follows the definition of the function.

```
data Sorted : List ℕ → Set where
  sort-[] : Sorted []
  sort-:: : ∀ {x} {xs} → ... → Sorted (x :: xs)
```

```
sorted-qs : ∀ {xs} → ∀ d → Sorted (quicksort xs d)
sorted-qs dom-[] = sort-[]
sorted-qs (dom-:: {x} {xs} p q) =
                    exp [x, xs, sorted-qs p, sorted-qs q]
```

Advantages of this Method

- Formalisations are easy to understand; close to functional programming style

- Separates logical and computational parts of a definition
  * Produces short type-theoretic functions
  * Allows the formalisation of partial functions
  * Simplifies formal verification

- Can be automatise

- Nested and mutually recursive functions present no problem (on type systems that support induction-recursion)

Nested Functions via the Graph

We define the graph, the domain and the function in a non-mutually dependent way (Bove 2009):

```
data _↓_ : ℕ → ℕ → Set where
  100< : ∀ n → 100 < n → n ↓ n - 10
  ≤100 : ∀ n x y → n ≤ 100 → n + 11 ↓ x → x ↓ y → n ↓ y
```

```
Dom91 : ℕ → Set
Dom91 n = ∃ (λ y → n ↓ y)
```

```
F91 : ∀ n → Dom91 n → ℕ
F91 n (y , _) = y
```

A Few Simple Results

$\texttt{unique-res} : \forall\ \texttt{n r l} \rightarrow \texttt{n} \downarrow \texttt{r} \rightarrow \texttt{n} \downarrow \texttt{l} \rightarrow \texttt{r} \equiv \texttt{l}$

$\texttt{dom-prf-ind} : \forall\ \texttt{n} \rightarrow \forall\ \texttt{p q} \rightarrow \texttt{F91 n p} \equiv \texttt{F91 n q}$

$\texttt{result} : \forall\ \texttt{n} \rightarrow \forall\ \texttt{p} \rightarrow \texttt{F91 n p} \equiv \texttt{proj}_1\ \texttt{p}$

$\texttt{im-}\!\downarrow\ : \forall\ \texttt{n} \rightarrow \forall\ \texttt{p} \rightarrow \texttt{n} \downarrow \texttt{F91 n p}$

$\texttt{res-}\!\downarrow\ : \forall\ \texttt{n} \rightarrow (\texttt{p} : \texttt{Dom91 n}) \rightarrow \ \texttt{n} \downarrow \texttt{proj}_1\ \texttt{p}$

Recursive Equations

```
eq-100< : ∀ n → ∀ p → 100 < n → F91 n p ≡ n - 10
```

```
eq-≤100 : ∀ n → ∀ p → n ≤ 100 →
              ∃ (λ p1 -> ∃ (λ p2 -> F91 n p ≡
                                     F91 (F91 (n + 11) p1) p2))
```

Graphs and Proofs

Step 1:

```
<result : ∀ {n} → ∀ p →  n < (F91 n p) + 11
<result (x , h) = ?
```

where `p : Dom91 n` and `h : n ↓ x`.

Graphs and Proofs

Step 1:

```
<result : ∀ {n} → ∀ p →  n < (F91 n p) + 11
<result (x , h) = ?
```

where `p : Dom91 n` and `h : n ↓ x`.

Step 2:

```
<result : ∀ {n} → ∀ p →  n < (F91 n p) + 11
<result (.(n - 10) , 100< n h) = exp1
<result (x , ≤100 n y .x h1 h2 h3) =
                         exp2 [<result (y , h2), <result (x , h3)]
```

where `exp1 : n < n - 10 + 11`

and `<result (y , h2) : n + 11 < F91 (n + 11) _ + 11`,

`<result (x , h3) : F91 (n + 11) _ < F91 (F91 (n + 11) _) _ + 11`

Advantages as Disadvantages

- ... basically as in the original domain predicate method
- As powerful as the original domain predicate method
- ... but a bit less direct
- However, *no need* for support for inductive-recursive definitions
- Needs some more case studies

Domain Predicates in Coq

In Coq, one can define (non-nested) recursive functions with a domain predicate of type

$$\mathsf{dom} \colon A \to \mathsf{Set}$$

in the same way as in Agda.

Domain Predicates in Coq

In Coq, one can define (non-nested) recursive functions with a domain predicate of type

$$\mathsf{dom} : A \rightarrow \mathsf{Set}$$

in the same way as in Agda.

```
Inductive dom : list Z -> Set :=
  | dom_nil : dom nil
  | dom_cons : forall (x:Z) (xs:list Z),
                     dom [ y | y <- xs , (Zlt_is_decidable x)] ->
                     dom [ y | y <- xs , (Zle_is_decidable x)] ->
                     dom (x::xs).
```

The Definition of Quicksort

```
Fixpoint quicksort (l : list Z) (H_dom : dom l)
                                {struct H_dom} : list Z :=
 match H_dom with
  | dom_nil => nil (A:=Z)
  | dom_cons x xs H_dom_lt H_dom_le =>
       quicksort [y | y <- xs, Zlt_is_decidable x] H_dom_lt ++
       x :: quicksort [y | y <- xs, Zle_is_decidable x] H_dom_le
  end.
```

```
Theorem everylist_in_dom : forall l, dom l.
```

```
Definition Quicksort l := quicksort l (everylist_in_dom l).
```

Problems with this Definition

- A domain of type $\mathsf{dom}\colon A \to \mathsf{Set}$ produces the wrong program after extraction!

Problems with this Definition

- A domain of type $\mathsf{dom}\colon A \rightarrow \mathsf{Set}$ produces the wrong program after extraction!

- In accordance with program extraction, the right type for the domain should be

$$\mathsf{dom}\colon A \rightarrow \mathsf{Prop}$$

Problems with this Definition

- A domain of type $\mathsf{dom}\colon A \to \mathsf{Set}$ produces the wrong program after extraction!

- In accordance with program extraction, the right type for the domain should be

$$\mathsf{dom}\colon A \to \mathsf{Prop}$$

- But then we cannot pattern match on the proof that the list belongs to the domain ...

Problems and Solution with this Definition

- A domain of type $\mathsf{dom}\colon A \to \mathsf{Set}$ produces the wrong program after extraction!

- In accordance with program extraction, the right type for the domain should be

$$\mathsf{dom}\colon A \to \mathsf{Prop}$$

- But then we cannot pattern match on the proof that the list belongs to the domain ...

- Solution: for each recursive call we need an *inversion* lemma showing that the proof arguments for the recursive calls can be deduced from the initial proof argument

New Domain Predicate in Coq

```
Inductive dom : list Z -> Prop :=
  | dom_nil : dom nil
  | dom_cons : forall (x:Z) (xs:list Z),
               dom [ y | y <- xs , (Zlt_is_decidable x)] ->
               dom [ y | y <- xs , (Zle_is_decidable x)] ->
               dom (x::xs).
```

New Domain Predicate in Coq

```
Inductive dom : list Z -> Prop :=
  | dom_nil : dom nil
  | dom_cons : forall (x:Z) (xs:list Z),
               dom [ y | y <- xs , (Zlt_is_decidable x)] ->
               dom [ y | y <- xs , (Zle_is_decidable x)] ->
               dom (x::xs).
```

```
Lemma dom_cons_inv_1 : forall l x xs, dom l ->
      l = x::xs -> dom [ y | y <- xs , (Zlt_is_decidable x) ].

Lemma dom_cons_inv_2 : forall l x xs, dom l ->
      l = x::xs -> dom [ y | y <- xs , (Zle_is_decidable x) ].
```

New Definition of Quicksort

```
Fixpoint quicksort (l : list Z) (H_dom : dom l)
                               {struct H_dom} : list Z :=
 match l as l0 return (l = l0 -> list Z) with
  | nil => fun _ : l = nil => nil
  | x :: xs =>
    fun H : l = x :: xs =>
        quicksort [y | y <- rest, Zlt_is_decidable x]
                  (dom_cons_inv_1 l x xs H_dom H) ++
        x :: quicksort [y | y <- rest, Zle_is_decidable x]
                       (dom_cons_inv_2 l x xs H_dom H)
  end (refl_equal l).


Theorem everylist_in_dom : forall l, dom l.
Definition Quicksort l := quicksort l (everylist_in_dom l).
```

Comments

(See Chapter 15 of Bertot and Castéran book on Coq (2004).)

- Inversion lemmas should be proved in such a way that their definition are seen as structurally smaller to the original proof argument

  (not by inversion but by pattern matching on the original proof argument, and returning a subproof)

- Their definition should also be transparent

- The standard induction principle for a predicate into Prop is usually not enough; we need the dependent version (maximal induction principle)

  ```
  Scheme dom_ind_dep := Induction for dom Sort Prop.
  ```

- Coq type system does not support inductive-recursive definitions, so nested recursion cannot be defined using domain predicates

The `Function` Command

(After work by Bertot and Balaa, and Barthe, Forest, Pichardie and Rusu.)

With the `Function` command one can define *total non-nested* functions:

- By structural recursion
- By giving a measure (into the Natural numbers) and proving that each recursive call is on smaller arguments
- By giving a well-founded relation and proving that each recursive call is on smaller arguments

The `Function` Command

(After work by Bertot and Balaa, and Barthe, Forest, Pichardie and Rusu.)

With the `Function` command one can define *total non-nested* functions:

- By structural recursion
- By giving a measure (into the Natural numbers) and proving that each recursive call is on smaller arguments
- By giving a well-founded relation and proving that each recursive call is on smaller arguments

It generates an induction principle that follows the definition of the function.

In the back, the graph is generated.

quicksort using Function

```
Function quicksort (l:list Z) {measure length} : list Z :=
  match l with
      nil => nil
  | x::xs  => let (ll,lg) := split x xs
                in quicksort ll ++ x :: quicksort lg
  end.
```

quicksort using Function

```
Function quicksort (l:list Z) {measure length} : list Z :=
  match l with
      nil => nil
  | x::xs  => let (ll,lg) := split x xs
                  in quicksort ll ++ x :: quicksort lg
  end.
```

Alternatively:

```
Definition lenR (l1 l2 : list Z) : Prop := length l1 < length l2.

Function quicksort (l:list Z) {wf lenR} : list Z :=
....
```

(In addition, we need to provide a proof that `lenR` is well-founded.)

Proof Obligations

We are left with 2 proof obligations:

```
______________________________________(1/2)
forall (l : list Z) (x : Z) (xs : list Z),
       l = x :: xs ->
       forall ll lg : list Z, split x xs = (ll, lg) ->
              length lg < length (x :: xs)


______________________________________(2/2)
forall (l : list Z) (x : Z) (xs : list Z),
       l = x :: xs ->
       forall ll lg : list Z, split x xs = (ll, lg) ->
              length ll < length (x :: xs)
```

Induction Principle

```
quicksort_ind =
fun P : list Z -> list Z -> Prop => quicksort_rect P
     : forall P : list Z -> list Z -> Prop,
       (forall l : list Z, l = nil -> P nil nil) ->
       (forall (l : list Z) (x : Z) (xs : list Z),
        l = x :: xs ->
        forall ll lg : list Z,
        split x xs = (ll, lg) ->
        P ll (quicksort ll) ->
        P lg (quicksort lg) ->
        P (x :: xs) (quicksort ll ++ x :: quicksort lg)) ->
       forall l : list Z, P l (quicksort l)
```

to be used with the tactic `functional induction`.

Function Package in Isabelle/HOL

(By Krauss, based on work by Slind.)

From the specification of the function the functional package:

- Extracts the recursive calls
- Produces the graph of the function
- Defines the function in Isabelle
- Defines the domain of the function
- Produce the recursive equations
- Produces an induction principle that follows the definition of the function

McCarthy $f91$ Function

Specification of the function given by the user:

```
fun f91 :: "nat => nat"
where
    "f91 n = if 100 < n then n - 10 else f91 (f91 (n + 11))"
```

McCarthy $f91$ Function

Specification of the function given by the user:

```
fun f91 :: "nat => nat"
where
    "f91 n = if 100 < n then n - 10 else f91 (f91 (n + 11))"
```

Recursive calls and their contexts are extracted:

$$\sim (100 < n) \rightsquigarrow n + 11 \qquad\qquad \sim (100 < n) \rightsquigarrow f91(n + 11)$$

McCarthy $f91$ Function

Specification of the function given by the user:

```
fun f91 :: "nat => nat"
where
    "f91 n = if 100 < n then n - 10 else f91 (f91 (n + 11))"
```

Recursive calls and their contexts are extracted:

$$\sim (100 < n) \leadsto n + 11 \qquad\qquad \sim (100 < n) \leadsto f91(n + 11)$$

For all h, the *graph* G is defined:

$$\frac{\begin{array}{c}\sim (100 < n) \Rightarrow (n + 11, h(n + 11)) \in G \\ \sim (100 < n) \Rightarrow (h(n + 11), h(h(n + 11))) \in G\end{array}}{(n, \textsf{if } 100 < n \textsf{ then } n - 10 \textsf{ else } h(h(n + 11))) \in G}$$

McCarthy $f91$ Function (Cont.)

The *function* is defined using HOL *definite description operator*:

$$f91 = \lambda x.\ THE\ y.\ (x, y) \in G$$

That is, the function is defined to take the value given by the graph, whenever the value exists and is unique.

Otherwise, the value of $f91$ is unspecified.

McCarthy $f91$ Function (Cont.)

The *function* is defined using HOL *definite description operator*:

$$f91 = \lambda x.\ THE\ y.\ (x, y) \in G$$

That is, the function is defined to take the value given by the graph, whenever the value exists and is unique.

Otherwise, the value of $f91$ is unspecified.

The *domain* D is as in the domain predicate method (though formulated in a different way):

$$\frac{\sim (100 < n) \Rightarrow (n + 11) \in D \qquad \sim (100 < n) \Rightarrow f91(n + 11) \in D}{n \in D}$$

McCarthy $f91$ Function (Cont.)

It should be proved that the graph G actually defines a function on D:

$$n \in D \Rightarrow \exists! y.\ (x, y) \in G$$

McCarthy $f91$ Function (Cont.)

It should be proved that the graph G actually defines a function on D:

$$n \in D \Rightarrow \exists! y.\ (x, y) \in G$$

The *recursive equation* is now guarded by a domain condition:

$$n \in D \Rightarrow f91\ n = \textsf{if}\ 100 < n\ \textsf{then}\ n - 10\ \textsf{else}\ f91(f91(n + 11))$$

McCarthy $f91$ Function (Cont.)

It should be proved that the graph G actually defines a function on D:

$$n \in D \Rightarrow \exists! y.\ (x, y) \in G$$

The *recursive equation* is now guarded by a domain condition:

$$n \in D \Rightarrow f91\ n = \mathsf{if}\ 100 < n\ \mathsf{then}\ n - 10\ \mathsf{else}\ f91(f91(n + 11))$$

The *induction principle* follows the definition of the function:

$$\frac{\forall n.n \in D \Rightarrow (\sim(100 < n) \Rightarrow P(n + 11)) \Rightarrow (\sim(100 < n) \Rightarrow P(f91(n + 11))) \Rightarrow Pn}{n \in D \Rightarrow Pn}$$

Function Package and Partiality

To help proving that the function is total, a *(nested) termination rule* is provided:

$$\frac{\mathsf{wf}\ R \qquad \sim(100 < n) \Rightarrow (n+11, n) \in R \qquad \sim(100 < n) \Rightarrow n+11 \in D \Rightarrow (f91(n+11), n) \in R}{\forall n.\ n \in D}$$

Function Package and Partiality

To help proving that the function is total, a *(nested) termination rule* is provided:

$$\frac{\mathsf{wf}\ R \qquad \sim(100 < n) \Rightarrow (n + 11, n) \in R \qquad \sim(100 < n) \Rightarrow n + 11 \in D \Rightarrow (f91(n + 11), n) \in R}{\forall n.\ n \in D}$$

If the functions has been proved total, then the domain condition in the recursive equations and in the induction principle can be removed.

This cannot be done neither in Agda nor in Coq!

Comments

- As shown, the package works fine with nested functions

- To deal with higher order functions, one can provide the system with *congruence rules*

  For example, for the map function we have

$$\frac{xs = ys \qquad x \in xs \Rightarrow f\ x = g\ x}{\texttt{map}\ f\ xs = \texttt{map}\ g\ ys}$$

  Then, the definition of the function

```
mirror (Node a ts) = Node a (map mirror (rev ts))
```

  produces the right domain condition

$$t \in (\texttt{rev}\ ts) \Rightarrow \texttt{mirror}\ t$$

- Similar for evaluation order

The PROGRAM Command (Sozeau)

Allows writing fully specified programs in a simple way.

Input terms are Coq term, but are typed in an weaker system call *Russell* which does not require terms to contain proofs.

Terms are then interpreted into Coq.

This process may produce proof obligations which need to be resolved to create the final term.

The `PROGRAM` Command (Sozeau)

Allows writing fully specified programs in a simple way.

Input terms are Coq term, but are typed in an weaker system call *Russell* which does not require terms to contain proofs.

Terms are then interpreted into Coq.

This process may produce proof obligations which need to be resolved to create the final term.

Main distinction:

$$\frac{\Gamma \vdash t\colon \{x\colon T \mid P\}}{\Gamma \vdash t\colon T} \qquad \frac{\Gamma \vdash t\colon T \qquad \Gamma \vdash P[t/x]}{\Gamma \vdash t\colon \{x\colon T \mid P\}}$$

Example PROGRAM: head

```
Program Definition head : { xs : list nat | xs <> [] } -> nat :=
  fun xs => match xs with
            | hd::tl => hd
            | [] => !
            end.
```

Generates the proof obligation

```
head_obligation_1
     : forall xs : {xs : list nat | xs <> []},
       let filtered_var := `xs in [] = filtered_var -> False
```

which is proved automatically.

Example PROGRAM: head

```
Program Definition ex : nat := head [6 ; 9].
ex has type-checked, generating 1 obligation(s)
Solving obligations automatically...
ex_obligation_1 is defined
No more obligations remaining
ex is defined

Check ex_obligation_1.
ex_obligation_1
     : [6; 9] <> []

Eval compute in ex.
     = 6
     : nat
```

Example PROGRAM: Nested Recursion

```
Program Fixpoint foo (n : nat) {measure id} : { m : nat | m <= n } :=
  match n with
  | 0 => 0
  | S p => foo (foo p)
  end.
```

Generates the obligations:

1. $\forall n. 0 = n \rightarrow 0 \leqslant 0$ (proved automatically)

2. $h_1 : \forall n. \forall p. \mathsf{S}\ p = n \rightarrow p < n$ (proved automatically)

3. $h_2 : \forall n. \forall p. \mathsf{S}\ p = n \rightarrow \mathsf{foo}\ (\mathsf{exists}\ p\ h_1) < n$

4. $\forall n. \forall p. \mathsf{S}\ p = n \rightarrow \mathsf{foo}\ (\mathsf{exists}\ (\mathsf{foo}\ (\mathsf{exists}\ p\ h_1))\ h_2) \leqslant n$

Sized Types: MiniAgda (Abel)

MiniAgda is an experimental prototype which implements a dependently typed core language with sized types.

Sizes can be seen as the height of the tree representing the structure of an element.

The idea is to annotate types with a size index representing the exact size of the element or an upper bound of it.

For recursive calls, the type system should check that the size of the argument decreases.

Sizes are irrelevant in the terms but not in the types.

Hence, types can depend on sizes but sizes should not influence the result of a function.

Example MiniAgda: `foo`

We have

- `$`: the successor function on sizes
- `#`: infinite size
- a size pattern $i > j$

Example MiniAgda: `foo`

We have

- `$`: the successor function on sizes
- `#`: infinite size
- a size pattern $i > j$

```
sized data Nat : Size -> Set
{ zero : [i : Size] -> Nat $i
; succ : [i : Size] -> Nat i -> Nat $i
}

fun foo : [i : Size] -> Nat i -> Nat i
{ foo i (zero (i > j))   = zero j
; foo i (succ (i > j) n) = foo j (foo j n)
}
```

More About Sized Types

- Sized types are especially good at higher-order functions
  (these functions are usually a problem...)

- Not quite ready to use in practice

- Listen to Andreas Abel on July 15th at PAR-10

- In the Coq community: Barthe, Gregoire and Riba
  A tutorial on type-based termination LerNet 2008, LNCS 5520

Thanks for listening!

And come to PAR-10 on July 15th to hear what is going on in partiality and recursion!