10 Years of Partiality and General Recursion in Type Theory

Ana Bove
Chalmers University of Technology

DTP’10 – July 9th 2010
Claims and Disclaims

\textit{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:

\[
\begin{align*}
tail : \{A : \text{Set}\} & \rightarrow \text{List } A \rightarrow \text{List } A \\
tail \; [] & = [] \\
tail \; (x :: \; xs) & = xs
\end{align*}
\]
Partial Functions

Functions not “defined” on a certain argument are not that problematic.

Some common solutions:

**Un-interesting result:**

\[
\text{tail} : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{List } A \\
\text{tail } [] = [] \\
\text{tail } (x :: xs) = xs
\]

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:**

\[
\text{tail} : \{A : \text{Set}\} \rightarrow \text{List A} \rightarrow \text{List A}
\]

\[
\text{tail} [] = []
\]

\[
\text{tail} (x :: \text{xs}) = \text{xs}
\]

What would we return for \text{head}?

**Maybe result:**

\[
\text{tail} : \{A : \text{Set}\} \rightarrow \text{List A} \rightarrow \text{Maybe (List A)}
\]

\[
\text{tail} [] = \text{nothing}
\]

\[
\text{tail} (x :: \text{xs}) = \text{just xs}
\]
Partial Functions (Cont.)

Restricted domain:

```haskell
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:

```haskell
-- 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 \ldots

Some methods to deal with \textit{(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 $A$:

\[
\begin{align*}
    a : A \quad (x : A) &\rightarrow x < a \rightarrow \text{Acc}(A, <, x) \\
    \text{Acc}(A, <, a) \quad (x : A) &\rightarrow \text{Acc}(A, <, x) \rightarrow ((y : A) \rightarrow y < x \rightarrow P(y)) \rightarrow P(x)
\end{align*}
\]
Well-Founded Recursion via Acc

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

$$
\frac{a : A \quad (x : A) \rightarrow x < a \rightarrow \text{Acc}(A, <, x)}{	ext{Acc}(A, <, a)}
$$

$$
\frac{\text{Acc}(A, <, a) \quad (x : A) \rightarrow \text{Acc}(A, <, x) \rightarrow ((y : A) \rightarrow y < x \rightarrow P(y)) \rightarrow P(x)}{	ext{P}(a)}
$$

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 : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\begin{align*}
ack 0 m &= \text{suc} m \\
ack (\text{suc} n) 0 &= \ack n 1 \\
ack (\text{suc} n) (\text{suc} m) &= \ack n (\ack (\text{suc} n) m)
\end{align*}

merge : List \mathbb{N} \to List \mathbb{N} \to List \mathbb{N}
\begin{align*}
\text{merge} \; [] \; ys &= ys \\
\text{merge} \; xs \; [] &= xs \\
\text{merge} \; (x :: \; xs) \; (y :: \; ys) &= \text{if} \; (x < y) \\
& \quad \text{then} \; (x :: \; \text{merge} \; xs \; (y :: \; ys)) \\
& \quad \text{else} \; (y :: \; \text{merge} \; (x :: \; xs) \; ys)
\end{align*}
Smarter Termination Checkers (Cont.)

\[ f : \{A : \text{Set}\} \rightarrow \text{List} A \rightarrow \text{List} A \rightarrow \text{List} A \]
\[ f \left[\right] \ y s = \left[\right] \]
\[ f \left(x ::\ x s\right) \ y s = f \ y s \ x s \]

\[ g : \{A : \text{Set}\} \rightarrow \text{List} A \rightarrow \text{List} A \rightarrow \text{List} A \]
\[ g \left[\right] = \left[\right] \]
\[ g \left(x ::\ []\right) = \left[\right] \]
\[ g \left(x ::\ y ::\ x s\right) = g \left(x ::\ x s\right) \]}
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:

\[
\text{all-dom} : \forall \; \text{xs} \rightarrow \text{dom} \; \text{xs}
\]

\[
\text{Quicksort} : \text{List} \; \mathbb{N} \rightarrow \text{List} \; \mathbb{N}
\]

\[
\text{Quicksort} \; \text{xs} = \text{quicksort} \; \text{xs} \; (\text{all-dom} \; \text{xs})
\]
Domain Predicates and Partiality

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

\[ \text{all-dom : } \forall \ xs \rightarrow \text{dom } xs \]

QuickSort : List \( \mathbb{N} \) \rightarrow List \( \mathbb{N} \)
QuickSort \( xs \) = quicksort \( xs \) (all-dom \( xs \))

But we can still talk about partial functions:

\[
\text{data dom-f : } \mathbb{N} \rightarrow \text{Set where} \\
\quad \text{dom-f-1 : dom-f } 1 \\
\quad \text{dom-f-s : } \forall \{n\} \rightarrow \text{dom-f (suc (suc n)) } \rightarrow \text{dom-f (suc (suc n))} \\
\]

\[
\text{f : } \forall \ n \rightarrow \text{dom-f } n \rightarrow \mathbb{N} \\
\quad \text{f .1 dom-f-1 = 0} \\
\quad \text{f (suc (suc n)) (dom-f-s p) = f (suc (suc n)) p}
\]
Domain Predicates and Nested Recursion

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

\[
\text{mutual}
\]

\[
data \text{dom91} : \mathbb{N} \rightarrow \text{Set}
\]

\[
data \text{dom100} : \forall \{n\} \rightarrow 100 < n \rightarrow \text{dom91} n
\]

\[
data \text{dom100} \leq 100 : \forall \{n\} \rightarrow n \leq 100 \rightarrow (p : \text{dom91} (n + 11)) \rightarrow \text{dom91} (f91 (n + 11) p) \rightarrow \text{dom91} n
\]

\[
f91 : \forall n \rightarrow \text{dom91} n \rightarrow \mathbb{N}
\]

\[
f91 n (\text{dom100} < h) = n - 10
\]

\[
f91 n (\text{dom100} \leq 100 h p q) = f91 (f91 (n + 11) p) q
\]
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 \(\mathbb{N}\) → Set where
  sort-[] : Sorted []
  sort-:: : \(\forall\) \{x\} {xs} → ... → Sorted (x :: xs)

sorted-qs : \(\forall\) {xs} → \(\forall\) 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):

\[
data \downarrow \downarrow : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set} \quad \text{where} \\
100 < : \forall n \rightarrow 100 < n \rightarrow n \downarrow n - 10 \\
\leq 100 : \forall n \times y \rightarrow n \leq 100 \rightarrow n + 11 \downarrow x \rightarrow x \downarrow y \rightarrow n \downarrow y
\]

\[\text{Dom91} : \mathbb{N} \rightarrow \text{Set}\]
\[\text{Dom91} n = \exists (\lambda y \rightarrow n \downarrow y)\]

\[\text{F91} : \forall n \rightarrow \text{Dom91} n \rightarrow \mathbb{N}\]
\[\text{F91} n (y , \_) = y\]
A Few Simple Results

unique-res : \( \forall \, n \, r \, l \rightarrow n \downarrow r \rightarrow n \downarrow l \rightarrow r \equiv l \)

dom-prf-ind : \( \forall \, n \rightarrow \forall \, p \, q \rightarrow F91 \, n \, p \equiv F91 \, n \, q \)

result : \( \forall \, n \rightarrow \forall \, p \rightarrow F91 \, n \, p \equiv \text{proj}_1 \, p \)

im-\downarrow : \( \forall \, n \rightarrow \forall \, p \rightarrow n \downarrow F91 \, n \, p \)

res-\downarrow : \( \forall \, n \rightarrow (p : \text{Dom}91 \, n) \rightarrow n \downarrow \text{proj}_1 \, p \)
Recursive Equations

\[ eq-100< : \forall n \rightarrow \forall p \rightarrow 100 < n \rightarrow F91\ n\ p \equiv n - 10 \]

\[ eq-\leq100 : \forall n \rightarrow \forall p \rightarrow n \leq 100 \rightarrow \exists (\lambda p1 \rightarrow \exists (\lambda p2 \rightarrow F91\ n\ p \equiv F91\ (F91\ (n + 11)\ p1)\ p2)) \]
Graphs and Proofs

Step 1:

\[ \text{result} : \forall \{n\} \rightarrow \forall p \rightarrow n < (F91 n p) + 11 \]

\[ \text{result} (x , h) = ? \]

where \( p : \text{Dom91} n \) and \( h : n \downarrow x \).
Graphs and Proofs

Step 1:

\[
<result : \forall \{n\} \rightarrow \forall p \rightarrow n < (F91 n p) + 11
\]

\[
<result (x , h) = ?
\]

where \( p : \text{Dom91} n \) and \( h : n \downarrow x \).

Step 2:

\[
<result : \forall \{n\} \rightarrow \forall p \rightarrow n < (F91 n p) + 11
\]

\[
<result (.\,(n - 10) , 100< n h) = \text{exp1}
\]

\[
<result (x , \leq 100 n y .x h1 h2 h3) = \text{exp2} [<result (y , h2), <result (x , h3)]
\]

where \( \text{exp1} : n < n - 10 + 11 \)

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

\[
\text{dom} : A \to \text{Set}
\]

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

\[ \text{dom} : A \rightarrow \text{Set} \]

in the same way as in Agda.

```
Inductive dom : list Z \rightarrow \text{Set} :=
| dom_nil : dom nil
| dom_cons : \forall (x:Z) (xs:list Z),
   dom [ y | y <- xs , (Zlt_is_decidable x)] \rightarrow
   dom [ y | y <- xs , (Zle_is_decidable x)] \rightarrow
   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 \( \text{dom}: A \rightarrow \text{Set} \) produces the wrong program after extraction!
Problems with this Definition

• A domain of type $\text{dom}: A \rightarrow \text{Set}$ produces the wrong program after extraction!

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

$$
\text{dom}: A \rightarrow \text{Prop}
$$
Problems with this Definition

- A domain of type \( \text{dom}: A \rightarrow \text{Set} \) produces the wrong program after extraction!

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

  \[
  \text{dom}: A \rightarrow \text{Prop}
  \]

- 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 \( \text{dom}: A \rightarrow \text{Set} \) produces the wrong program after extraction!

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

\[
\text{dom}: A \rightarrow \text{Prop}
\]

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

- Solution: for each recursive call we need an \textit{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 =>
    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 \texttt{Prop} is usually not enough; we need the dependent version (maximal induction principle)

\texttt{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)
\[
\forall (l : \text{list } Z) \ (x : Z) \ (xs : \text{list } Z), \quad
\begin{align*}
\text{if } l &= x :: xs \rightarrow \\
&\forall ll \ lg : \text{list } Z, \ \text{split x xs} = (ll, lg) \rightarrow \\
&\text{length lg} < \text{length } (x :: xs)
\end{align*}
\]

(2/2)
\[
\forall (l : \text{list } Z) \ (x : Z) \ (xs : \text{list } Z), \quad
\begin{align*}
\text{if } l &= x :: xs \rightarrow \\
&\forall ll \ lg : \text{list } Z, \ \text{split x xs} = (ll, lg) \rightarrow \\
&\text{length ll} < \text{length } (x :: xs)
\end{align*}
\]
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
McCarty $f_{91}$ Function

*Specification* of the function given by the user:

```ml
fun f91 :: "nat => nat"
where
  "f91 n = if 100 < n then n - 10 else f91 (f91 (n + 11))"
```
McCarty $f_{91}$ Function

*Specification* of the function given by the user:

```plaintext
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:

\[
\sim (100 < n) \leadsto n + 11 \quad \sim (100 < n) \leadsto f91(n + 11)
\]
**McCarthy f91 Function**

*Specification* of the function given by the user:

```plaintext
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:

\[
\sim (100 < n) \leadsto n + 11 \quad \sim (100 < n) \leadsto f91(n + 11)
\]

For all \( h \), the *graph* \( G \) is defined:

\[
\begin{align*}
\sim (100 < n) & \Rightarrow (n + 11, h(n + 11)) \in G \\
\sim (100 < n) & \Rightarrow (h(n + 11), h(h(n + 11))) \in G \\
(n, \text{if } 100 < n \text{ then } n - 10 \text{ else } h(h(n + 11))) & \in G
\end{align*}
\]
McCarthy \( f_{91} \) Function (Cont.)

The function is defined using HOL definite description operator:

\[
f_{91} = \lambda x. \text{THE } y. (x, y) \in G
\]

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 \( f_{91} \) is unspecified.
McCarthy $f91$ Function (Cont.)

The function is defined using HOL definite description operator:

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

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):

$$\sim (100 < n) \Rightarrow (n + 11) \in D \quad \sim (100 < n) \Rightarrow f91(n + 11) \in D$$

$$n \in D$$
**McCarthy $f^{91}$ 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 $f_{91}$ 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:

$$n \in D \Rightarrow f_{91} n = \text{if } 100 < n \text{ then } n - 10 \text{ else } f_{91}(f_{91}(n + 11))$$
**McCarthy $f_{91}$ 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:

$$n \in D \Rightarrow f_{91} n = \text{if } 100 < n \text{ then } n - 10 \text{ else } f_{91}(f_{91}(n + 11))$$

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

$$\forall n. n \in D \Rightarrow (\sim (100 < n) \Rightarrow P(n + 11)) \Rightarrow (\sim (100 < n) \Rightarrow P(f_{91}(n + 11))) \Rightarrow Pn$$

$$n \in D \Rightarrow Pn$$
To help proving that the function is total, a (nested) termination rule is provided:

\[
\begin{align*}
\text{wf } R & \quad \sim (100 < n) \Rightarrow (n + 11, n) \in R \\
\sim (100 < n) \Rightarrow n + 11 \in D \Rightarrow (f91(n + 11), n) \in R \\
\forall n. \ n \in D
\end{align*}
\]
To help proving that the function is total, a *(nested) termination rule* is provided:

\[
\begin{align*}
\text{wf } R & \quad \sim (100 < n) \Rightarrow (n + 11, n) \in R \\
\sim (100 < n) & \Rightarrow n + 11 \in D \Rightarrow (f 91(n + 11), n) \in R \\
\forall n. \ n \in D
\end{align*}
\]

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

\[
\begin{align*}
xs = ys & \quad x \in xs \Rightarrow f x = g x \\
\map f xs = \map g ys
\end{align*}
\]

Then, the definition of the function

\[
\text{mirror (Node a ts)} = \text{Node a (map mirror (rev ts))}
\]

produces the right domain condition

\[
t \in (\text{rev ts}) \Rightarrow \text{mirror t}
\]

• 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 a 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:

\[
\begin{align*}
\Gamma \vdash t : \{x : T \mid P\} & \quad \Gamma \vdash t : T & \quad \Gamma \vdash P[t/x] \\
\Gamma \vdash t : T & \quad \Gamma \vdash t : \{x : T \mid P\}
\end{align*}
\]
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 := \texttt{`xs in [] = filtered_var \rightarrow False}\n
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. $\forall n.0 = n \rightarrow 0 \leq 0$ (proved automatically)

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

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

4. $\forall n.\forall p.\text{S } p = n \rightarrow \text{foo (exists } (\text{foo (exists } p \ h_1)) \ h_2) \leq n$
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!