Partiality and General Recursion in Type Theory
(old news actually :-)

Ana Bove

Initial Types Club

12th March 2020

(Lots of) Disclaims

- Will not talk much about functions not defined on an argument
- Will talk about solutions in AGDA (not in COQ or other implementations of type theory)
- Will not talk about all possible solutions in AGDA: see for example sized-types (Abel)
- Will not consider solutions using co-inductive types: see for example partiality modal (Capretta, ...)
- Will not consider recursion on co-inductive functions
- Code showed is probably not correct here and there
- Results presented here are not too recent

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?

Functions not Defined on an Argument

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
```

Functions not Defined on an Argument (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.)

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 an explicit *measure* and show it is *well-founded*.

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)
```

How to Formalise General Recursion?

Let us consider the `quicksort` example:

```
quicksort ::  List ℕ → List ℕ
quicksort [] = []
quicksort (x : xs) = quicksort (filter (< x) xs) ++
                     x : quicksort (filter (≥ x)) xs
```

It is easy to reason that it always terminates.

Accessibility Predicate

Let A be a set and $<$ a (well-founded) binary relation over A.

That is, there are no infinite $<$-chains starting from a:

$$\frac{a : A \qquad (x : A) \rightarrow x < a \rightarrow \mathrm{Acc}(A, <, x)}{\mathrm{Acc}(A, <, a)}$$

The induction principle `wfr` is a generalisation of the course-of-value induction to an arbitrary set A and well-founded relation $<$:

$$\frac{\begin{array}{c}\mathrm{Acc}(A, <, a) \\ (x : A) \rightarrow \mathrm{Acc}(A, <, x) \rightarrow ((y : A) \rightarrow y < x \rightarrow P(y)) \rightarrow P(x)\end{array}}{P(a)}$$

Well-Founded Recursion via Accessibility Predicate

Let

```
≺ : List ℕ → List ℕ → Set
allacc : ∀ xs → Acc (List ℕ, ≺ , xs)

prlt :  ∀ x → ∀ xs → filter (< x) xs ≺ x :: xs
prge :  ∀ x → ∀ xs → filter (≥ x) xs ≺ x :: xs
```

in

```
quicksort : List ℕ → List ℕ
quicksort xs = wfr (allacc xs) qs
  where qs : ∀ xs → (∀ ys → ys ≺ xs → List ℕ) → List ℕ
        qs [] h = []
        qs (x :: xs) h = h (filter (< x) xs) (prlt x xs) ++
                           x :: h (filter (≥ x) xs) (prge x xs)
```

Well-Founded Recursion via Accessibility Predicate (Cont.)

Representing general recursion with the accessibility predicate has some 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)

Domain Predicates (Bove/Capretta) in Agda

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

```
data dom : List ℕ → Set where
  dom-[] : dom []
  dom-:: : ∀ {x} {xs} → dom (filter (< x) xs) →
                         dom (filter (≥ x) xs) →
                         dom (x :: xs)
```

```
quicksort : ∀ xs → dom xs → List ℕ
quicksort [] dom-[] = []
quicksort (x :: xs) (dom-:: p q) =
                  quicksort (filter (< x) xs) p ++
                  x :: quicksort (filter (≥ x) xs) q
```

Domain Predicates on Total Functions

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

```
all-dom : ∀ xs → dom xs
all-dom xs = wfr ...

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

The definition of `all-dom` will have a similar structure than the definition of `quicksort` using the accessibility predicate.

Domain Predicates and Partial Functions

With these domains 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
```

```
zero : ℕ
zero = f 1 dom-f-1
```

How to Formalise Nested Recursion?

Let us consider McCarthy `f91` function:

```
f91 n | 100 < n = n - 10
f91 n | otherwise = f91 (f91 (n + 11))
```

Not immediate, but we can reason that it always terminates with the value:

- $n - 10$ if $100 < n$
- 91 if $100 \geq n$

Domain Predicates and Nested Recursion

Using the schema for induction-recursion definitions (Dybjer) we can define nested recursive functions:

```
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!

```
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 (x :: xs) (dom-:: p q) =
            let sqs-<  = sorted-qs (filter (< x) xs) p
                sqs-≥ = sorted-qs (filter (≥ x) xs) q
            in ...
```

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
- Could be automatise
- Nested and mutually recursive functions present no problem
  (hmm, is this actually true?)

Problem with Domain Predicates for Nested Recursion

Not all type theories support Dybjer's schema for simultaneous inductive-recursive definitions:

- Martin-Löf type theory: yes
- Calculus of Inductive Constructions: no

How can we solve this?

Solution: Use the Graph to Define the Domain

```
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 = ∃ ℕ (λ m → n ↓ m)
```

```
f91 : ∀ n → dom91 n → ℕ
f91 n ((m , _)) = m
```

Observe that there is no mutual dependency between the definitions!

Advantages of Using the Graph to Define the Domain

- Basically as in the method that uses domain predicates but proofs are a bit less direct: from the domain one moves to the graph where one actually does the induction
- Seems as powerful as the original domain predicate method (haven't done too many examples with it)
- No need for support for inductive-recursive definitions

How to Formalise Higher-Order Recursion?

Let us consider the following `Tree` datatype and a `mirror` function over it:

```
data Tree A =
  tree :: A → List (Tree A) → Tree A
```

```
mirror ::  Tree A → Tree A
mirror (tree a ts) = tree a (rev (map mirror ts))
```

The termination of the `mirror` function critically depends on how the higher-order map function uses its argument.

Intuitively easy to see it terminates in this case...

Formalisation of Higher-Order Recursion

Many methods are not really useful, in particular that based on domain predicates.

Some possible methods to use are:

- Some kind of well-founded recursion
- Type-based termination methods such as sized types
- `agda2atp` tool:
  - Encodes a first-order theories in AGDA using the formulae-as-types principles
  - Translates AGDA representation of first-order formulae into TPTP
  - Calls first-order theorem provers for first-order logic

Some Relevant Work on Other Systems

- `Function` command in COQ (Balaa, Bertot, Barthe, ...)
- `Program` package in COQ (Sozeau)
- `Equation` package in COQ (Sozeau)
- `Function` package in ISABELLE/HOL (Krauss)

See *Partiality and recursion in interactive theorem provers – an overview* by A. Bove, A. Krauss and M. Sozeau for more information (2012).