Using constructive type theory for reasoning about general recursive algorithms

Andrés Sicard-Ramírez

(with the co-supervision of Ana Bove and Peter Dybjer at Chalmers University of Technology, Sweden)

Universidad EAFIT, Colombia
Universidad de la República, Uruguay
LERnet grant-holder

LERnet summer school
Piriápolis, Uruguay
25 February - 1 March 2008

Goal and problems

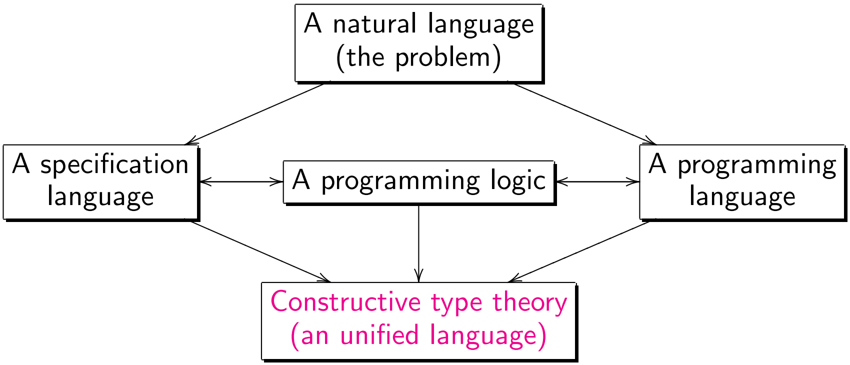
Goal

Our goal is to use constructive type theory for reasoning about general recursive functional programs and to explore possible connections between functional programming languages, proof assistants, and automatic theorem provers.

Problems

- To represent general recursive functions in constructive type theory
- To define a logical framework for reasonign about general recursive functional programs
- To implement the logical framework in the proof assistant Agda

Reasoning about programs: the languages



Constructive type theory (CTT)

$$
\text{CTT}\begin{cases}
\text{Dependent types} \\
\quad \text{(Types that depend on elements of others types)} \\
\text{Curry-Howard isomorphism} \begin{cases} a & : A \\ \text{proof} & : \text{proposition} \\ \text{program} & : \text{specification} \end{cases} \\
\text{Inductive definitions (data types, inductive predicates)} \\
\text{Notion of computation (reduction of programs to values)} \\
\text{Notion computational equality (conversion)}
\end{cases}
$$

Constructive type theory (CTT)

CTT

- Dependent types
  (Types that depend on elements of others types)
- Curry-Howard isomorphism

a	: A
proof	: proposition
program	: specification

- Inductive definitions (data types, inductive predicates)
- Notion of computation (reduction of programs to values)
- Notion computational equality (conversion)

$\Downarrow$

Correct programs by construction (strong specification) (integrated logic)	Programs verification (weak specification) (external logic)

Constructive type theory (CTT)

Example (strong specification for the greatest common divisor)

(Agda notation)

```
data ∃ (A : Set) (P : A -> Set) : Set where
  ∃-i : (witness : A) -> P witness -> ∃ A P


-- The relation of divisibility
data _||_ (m n : Nat) : Set where
  ||-i :  ∃ Nat (\k -> isTrue (n =N= (k * m))) -> m || n


gcd : (m n : Nat) ->
      ∃ r Nat ( r || m ∧ r || n ∧
                ((r' : Nat) -> r' || m -> r' || n -> r ≥ r')
              )
gcd = ...
```

Constructive type theory (CTT)

Example (a weak specification for the greatest common divisor)

```
gcd : Nat -> Nat -> Nat
gcd = ...

gcd-fst : (m n : Nat) -> gcd m n || m
gcd-fst = ...

gcd-ge : (m n r' : Nat) -> r' || m -> r' || n -> gcd m n ≥ r'
gcd-ge = ...
```

The limitation

Constructive type theory: a total functional programming language

> *"... I could not think of dealing with ... possibly non-terminating programs, before I had freed myself from the interpretation of propositions as types" (P. Martin-Löf, 1985, p. 184)*

- Consistency (simple minded consistency)
- Decidability of type-checking

The restriction: structural recursion

Recursive programs in which the recursive calls have structurally smaller arguments

The limitation

Example (structural recursion)

```
data Nat : Set where
  zero  : Nat
  suc   : Nat -> Nat

-- a primitive recursion function
_+_ : Nat -> Nat -> Nat
m + zero    = m
m + (suc n) = suc (m + n)

-- 'iterate' is a higher-order function that iterates a function
-- a certain number of times
iterate' : (A : Set) -> (f : A -> A) -> Nat -> A -> A
iterate' A f zero    x = x
iterate' A f (suc n) x = f (iterate' A f n x)

-- Other version of '_+_'
_+'_ : Nat -> Nat -> Nat
_+'_ = iterate' Nat suc
```

The limitation

Example (non-structural recursion)

```
-- The greatest common divisor using repeated subtraction
gcd : Nat -> Nat -> Nat
gcd zero    n       = n
gcd m       zero    = m
gcd (suc m) (suc n) = if m > n
                      then gcd (suc m - suc n)  suc n
                      else gcd suc m (suc n - suc m)
```

The limitation

The restriction: structural recursion

Recursive programs in which the recursive calls have structurally smaller arguments

Consequence

There is no direct way to represent general recursive programs in constructive type theory

Representation of general recursion in constructive type theory: some proposals

Inductive definition of domain predicates

(A. Bove and V. Capretta, 2005)

```
-- a general recursive function
f : A₁ -> ··· -> Aₙ -> B
f = ...

-- an inductive special-purpose accessible predicate
-- that characterizes the inputs on which the
-- function terminates
data fDom : A₁ -> ··· Aₙ -> Set where ...

-- structural recursive version on 'fDom' of 'f'
fIDP : (a₁ : A₁)···(aₙ : Aₙ)  -> fDom a₁ ... aₙ  -> B
fIDP = ...
```

Representation of general recursion in constructive type theory: some proposals

Example (Representation of 'gcd' using an inductive domain predicates)

From 'gcd' definition we have the inductive domain predicate

$$\frac{n : \mathrm{Nat}}{\mathrm{gcdDom}\ \mathit{zero}\ n} \qquad \frac{m : \mathrm{Nat}}{\mathrm{gcdDom}\ m\ \mathit{zero}}$$

$$\frac{m, n : \mathrm{Nat} \qquad \mathrm{suc}\ m > \mathrm{suc}\ n \qquad \mathrm{gcdDom}\ (\mathrm{suc}\ m - \mathrm{suc}\ n)(\mathrm{suc}\ n)}{\mathrm{gcdDom}\ (\mathrm{suc}\ m)(\mathrm{suc}\ n)}$$

$$\frac{m, n : \mathrm{Nat} \qquad \neg(\mathrm{suc}\ m > \mathrm{suc}\ n) \qquad \mathrm{gcdDom}\ (\mathrm{suc}\ m)(\mathrm{suc}\ n - \mathrm{suc}\ m)}{\mathrm{gcdDom}\ (\mathrm{suc}\ m)(\mathrm{suc}\ n)}$$

Representation of general recursion in constructive type theory: some proposals

Example (Representation of 'gcd' using an inductive domain predicates)

The implementation of the inductive predicate

```
data gcdDom : Nat -> Nat -> Set where
  gcdDom-c1 : (m : Nat) -> gcdDom m zero
  ...
  gcdDom-c4 : (m n : Nat) -> isFalse (suc m > suc n) ->
              gcdDom (suc m) (suc n - suc m) ->
              gcdDom (suc m)(suc n)
```

The definition of 'gcd' using structural recursive on 'gcdDom'

```
gcdIDP : (m n : Nat)  -> gcdDom m n -> Nat
gcdIDP a       zero    (gcdDom-c1 .a)           = a
...
gcdIDP (suc a) (suc b) (gcdDom-c4 .a .b p1 p2) =
  gcIDP (suc a) (suc b - suc a) p2
```

Representation of general recursion in constructive type theory: some proposals

Representation of functions as relations

```
-- a general recursive function
f : A₁ -> ··· -> Aₙ -> B
f = ...

-- an inductive relation that relates the 'n' input values
-- with the result value

-- it is necessary remember that the relation 'fR' must
-- satisfies the function's constraints
data fR : A₁ -> ··· -> Aₙ -> B -> Set where ...
```

Representation of general recursion in constructive type theory: some proposals

Example (Representation of 'gcd' as an inductive relation)

```
data gcdR : Nat -> Nat -> Nat -> Set where
  gcdR-c1 : (m : Nat) -> gcdR  m zero m
  ...
  gcdR-c4 : (m n v : Nat) -> isFalse (suc m > suc n)  ->
            gcdR (suc m) (suc n - suc m) v ->
            gcdR (suc m)(suc n) v
```

Our proposal: To use the Logical Theory of Constructions (LTC)

LTC as an unified language for reasoning about general recursive programs
(P. Dybjer, 1985, 1986, 1990)

$$\text{LTC} \begin{cases} \text{Constructive predicate logic with equality} \\ \text{Type-free functional programming language} \\ \text{Inductively defined predicates} \\ \text{Notion of computation} \\ \text{Notion computational equality} \end{cases}$$

Our proposal: To use the Logical Theory of Constructions (LTC)

LTC and CTT

LTC
- Constructive predicate logic with equality
- Type-free functional programming language
- Inductively defined predicates

CTT
- Dependent types
- Curry-Howard isomorphism
- Inductive definitions

LTC-CTT
- Notion of computation
- Notion computational equality

Our proposal: To use the Logical Theory of Constructions (LTC)

Some LTC's features

- Not Curry-Howard isomorphism
- Interpretation of types as inductive predicates
- We can define general recursive functions
- Proving that a function has a type amounts to proving its termination (P. Dybjer, 1985, 1986)
- LTC is at least as strong as Martin-Löf type theory (J. Smith, 1978, 1984)

The implementation: Using Agda as an logical framework

Example (Partial data types)

```
postulate D : Set

-- Partial booleans
postulate #true  : D
postulate #false : D

-- Partial natural numbers
postulate #zero : D
postulate #suc  : D -> D
```

The implementation: Using Agda as an logical framework

Example (Introduction rules for the terminating natural numbers 'Nat')

$$\frac{}{\mathsf{Nat}\ \#\mathit{zero}} \qquad \frac{\mathsf{Nat}\ n}{\mathsf{Nat}\ (\#\mathit{suc}\ n)}$$

```
-- The pure logical framework
postulate Nat  : D -> Set
postulate zero : Nat #zero
postulate suc  : (n : D) -> Nat n -> Nat (#suc n)

-- A mixed logical framework
data Nat : D -> Set where
  zero : Nat #zero
  suc  : (n : D) -> Nat n -> Nat #suc n)
```

The implementation: Using Agda as an logical framework

Example (Equality for the type 'D')

```
-- The pure logical framework
postulate _=D=_ : D -> D -> Set

postulate =D=-refl  : (d : D) -> d =D= d
postulate =D=-subst : (P : D -> Set){e₁ e₂ : D} -> e₁ =D= e₂ ->
                       P e₁ -> P e₂
```

Example (if-then-else)

```
postulate if_then_else_ : D -> D -> D -> D
postulate ite-axT : (e₁ e₂ : D) ->
                    if #true then e₁ else e₂ =D= e₁
postulate ite-axF : (e₁ e₂ : D) ->
                    if #false then e₁ else e₂ =D= e₂
```

The implementation: Using Agda as an logical framework

Example (Representation of 'gcd' on LTC)

```
postulate gcdLTC : D -> D -> D

postulate gcdLTC-ax : (m n : D) ->
             gcdLTC m n =D=
               if (n =#N= #zero)
               then m
               else (if (m =#N= #zero)
                     then n
                     else (if (m >D n )
                           then gcdLTC (m -D n) n
                           else gcdLTC m (n -D m)
                          )
                    )


gcdLTC-Nat : NatT (gcdLTC m n)
gcdLTC-Nat = ...
```

To-do list

- Theoretical work (CTT-LTC, extensions for LTC, etc)
- The implementation: Agda as an (mixed) logical framework (data types versus postulates, equalities, equality reasoning, etc.)
- LTC can be formalized as a first order theory with equality. To implement an external automatic theorem prover from Agda.
- ...