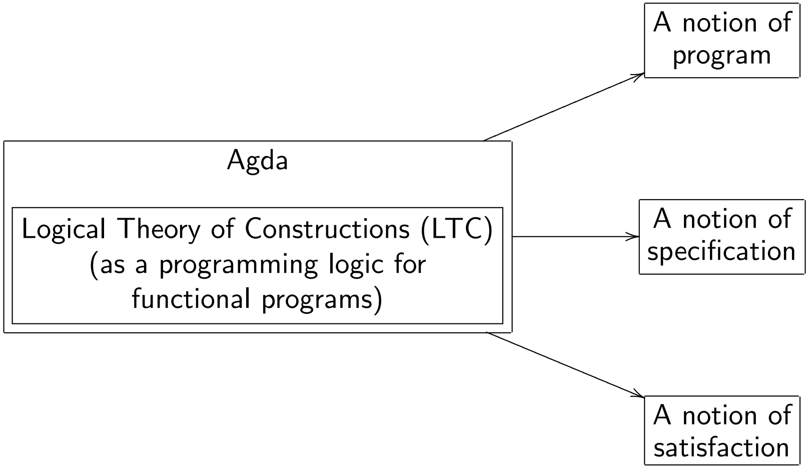
Using a logical theory of constructions for program verification

Andrés Sicard Ramírez
(joint work with Ana Bove and Peter Dybjer)

Universidad EAFIT, Colombia

AIM8, Gothenburg
30 May 2008

The idea


Agda
Logical Theory of Constructions (LTC)
(as a programming logic for
functional programs)
A notion of
program
A notion of
specification
A notion of
satisfaction

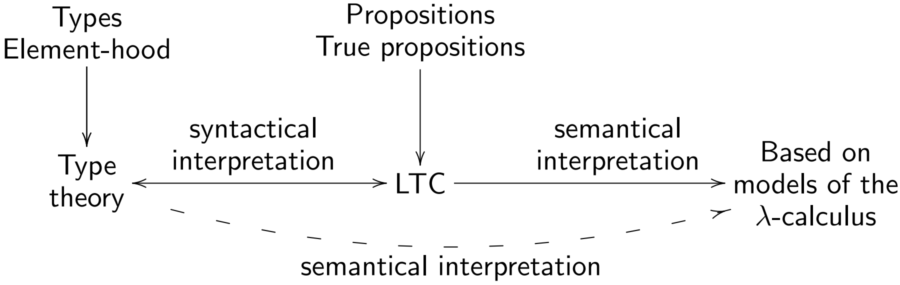
Historical background

Logical theory of constructions (LTC): original motivation

(P. Aczel 1974, 1980 and J. Smith 1978, 1984)

> *"The basic LTC framework is intended to be, at the informal level, the framework of ideas that are being used by Per Martin-Löf in his semantical explanations for ITT. Those explanations seem to treat the notions of proposition and truth as fundamental and use them to explain the notions of type and element-hood as used in ITT". (P. F. Mendler and P. Aczel, 1988, p. 393)*

Historical background (cont.)

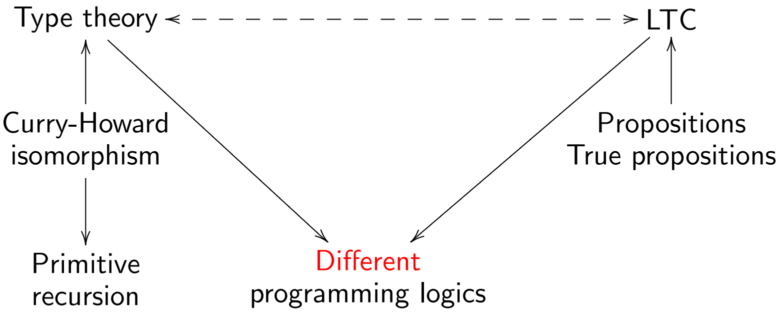


Foundational remark

> *"This will not mean that we consider the logical theory more fundamental than type theory. Of course, the logical theory also needs a semantical explanation and this can presumably not be given as easily as for the type theory in Martin-Löf." (J. Smith, 1984, p. 730-1)*

Why use LTC as a programming logic?

(P. Dybjer 1985, 1986, 1990)



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

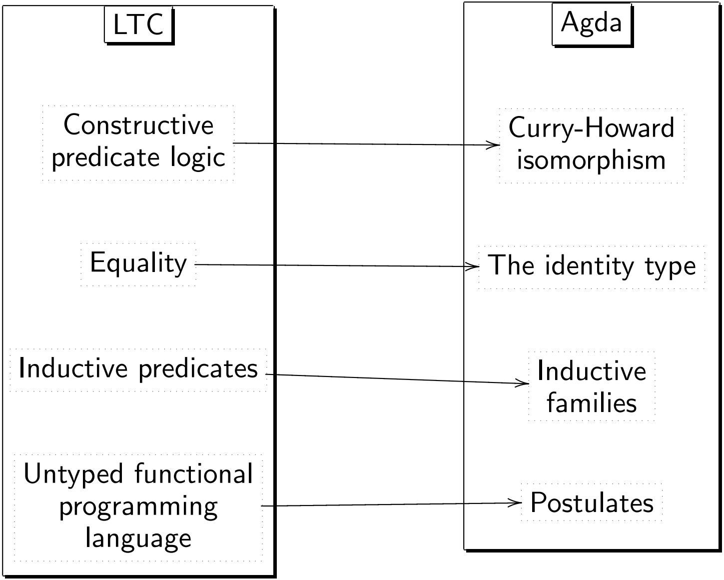
LTC as a programming logic

- A notion of program
- A notion of specification
- A notion of satisfaction

- Untyped functional programming language
- Constructive predicate logic with equality and inductive predicates
- Inference rules (logical rules, conversion rules, and inductive predicates rules)

Agda as a logical framework for LTC

A mixed logical framework approach



LTC's terms

A untyped functional programming language

D : an universal domain of terms
'Weak' types: Agda's simple type lambda calculus on D

$$\mathcal{T} ::= \mathtt{D} \mid \mathcal{T} \to \mathcal{T}$$
$$t ::= x \mid \backslash x \to t \mid t\ t \mid \mathit{consts}$$

Constants terms's approach

- Theoretical: fixed
- Practical: open

LTC's terms (cont.)

Constant terms

```
postulate

  -- The universal domain
  D : Set

  -- LTC booleans
  true#           : D
  false#          : D
  if#_then_else_ : D -> D -> D -> D

  -- LTC natural numbers
  zero# : D
  succ# : D -> D
  rec#  : D -> D -> (D -> D -> D) -> D

  -- LTC abstraction and application
  λ   : (D -> D) -> D
  _’_ : D -> D -> D
```

LTC's inductive predicates and propositions

LTC's inductive predicates

inductive predicates: type theory types for the LTC-programs

LTC's propositions

Constructive predicate logic with equality + inductive predicates

$$P ::= (\forall x)P \mid (\exists x)P \mid P \supset P \mid P \wedge P \mid P \vee P \mid \bot \mid t == t$$
$$\mid N(t) \text{ (natural numbers)}$$

LTC's inference rules: logical rules

- Logical constants: standard ones
- Equality rules

$$\frac{}{t == t} \qquad\qquad \frac{s == t \qquad P(s)}{P(t)}$$

```
-- The identity type
data _==_ {A : Set}(x : A) : A -> Set where
  ==-refl : x == x

==-subst : {A : Set}(P : A -> Set){x y : A} -> x == y ->
           P x -> P y
==-subst P ==-refl px = px
```

LTC's inference rules: natural numbers

Introduction rules

$$\frac{}{N(zero\#)} \qquad\qquad \frac{N(n)}{N(succ\#\ n)}$$

Elimination rule

$$\frac{N(n) \qquad P(zero\#) \qquad \begin{array}{c}[N(x), P(x)] \\ \vdots \\ P(succ\#\ x)\end{array}}{P(n)}\ (*)$$

(*) x must not occur free in any assumption on wich $P(succ\#\ x)$ depends other than $N(x)$ and $P(x)$

LTC's inference rules: natural numbers (cont.)

```
-- The natural numbers type

data N : D -> Set where
  zeroN : N zero#
  succN : (n : D) -> N n -> N (succ# n)

-- Induction principle on N   (elimination rule)

N-ind : (P : D -> Set) ->
        P zero# ->
        ({n : D} -> N n -> P n -> P (succ#  n)) ->
        {n : D} -> N n -> P n
N-ind P p0 h zeroN        = p0
N-ind P p0 h (succN n Nn) = h Nn  (N-ind P  p0  h Nn)
```

LTC's inference rules: conversion rules

```
postulate
    -- Conversion rules for booleans
    CB1 : (a : D){b : D} -> if# true# then a else b == a
    CB2 : {a : D}(b : D) -> if# false# then a else b == b

    -- Conversion rules for natural numbers
    CN1 : (a : D)(f : D -> D -> D) -> rec# zero# a f == a
    CN2 : (a n : D)(f : D -> D -> D) ->
          rec# (succ# n) a f == f n (rec# n a f)

    -- Conversion rule for the abstraction and the application
    beta : (f : D -> D)(a : D) -> (λ f) ' a == f a
```

Example

```
-- Recall we postulated
λ   : (D -> D) -> D
_'_ : D -> D -> D
beta : (f : D -> D)(a : D) -> (λ f) ' a == f a

-- non-terminating programs
ω : D
ω = λ(\x -> x ' x)

Ω : D
Ω = ω ' ω

-- a fixed point operator
fix : (D -> D) -> D
fix f = λ (\x -> f(x ' x)) ' λ (\x -> f(x ' x))
```

Example: the greatest common divisor using repeated subtraction

```
_-_ : D -> D -> D
eq : D -> D -> D
gt : D -> D -> D

postulate
    gcd : D -> D -> D

    -- first version
    Cgcd : (m n : D) ->
      gcd m n == if# (eq n zero#)
                    then m
                    else if# (eq m zero#)
                             then n
                             else if# (gt m n)
                                      then gcd (m - n)  n
                                      else gcd  m  (n - m)
```

Example: the greatest common divisor using repeated subtraction (cont.)

```
_-_ : D -> D -> D
eq : D -> D -> D
gt : D -> D -> D

postulate
    gcd : D -> D -> D

    -- second version
    Cgcd1 : (m : D) ->   gcd m          zero#      == m
    Cgcd2 : (n : D) ->   gcd zero#      n          == n
    Cgcd3 : (m n : D) -> gcd (succ# m) (succ# n) ==
              if# (gt (succ# m) (succ# n))
                then gcd ((succ# m) - (succ# n)) (succ# n)
                else gcd (succ# m) ((succ# n) - (succ# m))
```

Program verification on the logical theory of constructions

Example (the greatest common divisor using repeated subtraction)

Given the program to calculate the `gcd`, we want to prove

$$(\forall m, n \in N)(gcdP(m, n, (gcd\ m\ n))$$

where

$$\begin{aligned}
(\forall x \in A)B(x) &\equiv_{def} (\forall x)(A(x) \supset B(x)) \\
(\exists x \in A)B(x) &\equiv_{def} (\exists x)(A(x) \wedge B(x)) \\
a \mid b &\equiv_{def} (\exists k \in N)(b == k * a) \\
gcdP(m, n, r) &\equiv_{def} (r \mid m) \wedge \\
&\quad\ (r \mid n) \wedge \\
&\quad\ ((\forall r' \in N)(r' \mid m \wedge r' \mid n \supset r \geq r'))) \wedge \\
&\quad\ N(r)
\end{aligned}$$

Future work

- To strengthen the mixed logical framework approach (i.e. to use the primitive recursive functions of Agda)

```
nat2n# : Nat -> D

nat2n : (n : Nat) -> N (nat2n# n)

n#2nat : (d : D) -> N d -> Nat
```

- New Agda feature: foreign function interface for calling Haskell functions from Agda
- How we can combine our implementation with an automatic theorem prover?

Future work (cont.)

- LTC and others programming logics

	TT	LTC	LCF	...
Logic	constructive	constructive	classical	...
Logic	integrated	external	external	...
Recursion	primitive	general	general	...
Objects	total	partial	partial	...

- Termination properties on LTC (simple types)

$$
\begin{aligned}
a \in A &\equiv_{def} A(a) \\
b \in Bool &\equiv_{def} b == true\# \vee b == false\# \\
q \in A + B &\equiv_{def} (\exists x \in A)(q == inl\#\ x)) \vee (\exists x \in B)(q == inr\#\ x)) \\
f \in A \rightarrow B &\equiv_{def} (\exists b)((\forall x)(x \in A \supset b(x) \in B)) \wedge f == \lambda(b))
\end{aligned}
$$

Final remarks

The logical theory of constructions is an appropriate constructive programming logic for reasoning about general recursive functional programs:

- It has not the limitations due to the Curry-Howard isomorphism, that is to say, we can define general recursive functions as their Haskell-like versions.
- Proving that a program has a type (i.e. its value belongs to a simple type) amounts to proving its termination
- It is at least as strong as Martin-Löf type theory

References I

[Acz77] Peter Aczel.

The strength of Martin-Löf's intuitionistic type theory with one universe.

In *Proc. of the symposium on mathematical logic (Oulu, 1974)*, Report No. 2, Department of Philosopy, University of Helsinki, Helsinki, pages 1–32, 1977.

[Acz80] Peter Aczel.

Frege structures and the notion of proposition, truth and set.

In *The Kleene Simposium*, pages 31–59. Amsterdan: North-Holland, 1980.

[Dyb85] Peter Dybjer.

Program verification in a logical theory of constructions.

In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 210 of *LNCS*, pages 334–349, 1985.

References II

[Dyb86] Peter Dybjer.
Program verification in a logical theory of constructions.
Technical report, Programming Methodology Group Report 26, University of Göteborg and Chalmers University of Technology, 1986.
Revision of [Dyb85].

[Dyb90] Peter Dybjer.
Comparing integrated and external logics of functional programs.
Science of Computer Programming, 14:59–79, 1990.

[MA88] Paul F. Mendler and Peter Aczel.
The notion of a framework and a framework for LTC.
In *Proc. of the Third Annual Symposium on Logic in Computer Science (LICS '88)*, pages 392–399. IEEE, 1988.

References III

[ML82] Per Martin-Löf.

Constructive mathematics and computer programming.

In L. J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Logic, Methodology and Philosophy of Science VI (1979)*, pages 153–175. Amsterdam: North-Holland Publishing Company, 1982.

[ML85] Per Martin-Löf.

Constructive mathematics and computer programming.

In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice/Hall International, 1985.

Reprinted from [ML82] with a short discussion added.

References IV

[Smi78] Jan Smith.
On the relation between a type theoretic and a logic formulation of the theory of constructions.
PhD thesis, Chalmers University of Technology and Göteborg University, Department of Mathematics, 1978.

[Smi84] Jan Smith.
An interpretation of Martin-Löf's type theory in a type-free theory of propositions.
The Journal of Symbolic Logic, 49(3):730–753, 1984.