Embedding a logical theory of constructions in Agda

Andrés Sicard-Ramírez[1]
joint work with Ana Bove[2] and Peter Dybjer[2]

[1]EAFIT University, Colombia

[2]Chalmers University of Technology, Sweden

PLPV'09
Savannah, Georgia

Introduction

Motivation - Long term goal

To use a proof assistant for verifying programs written in a standard functional language such as Haskell.

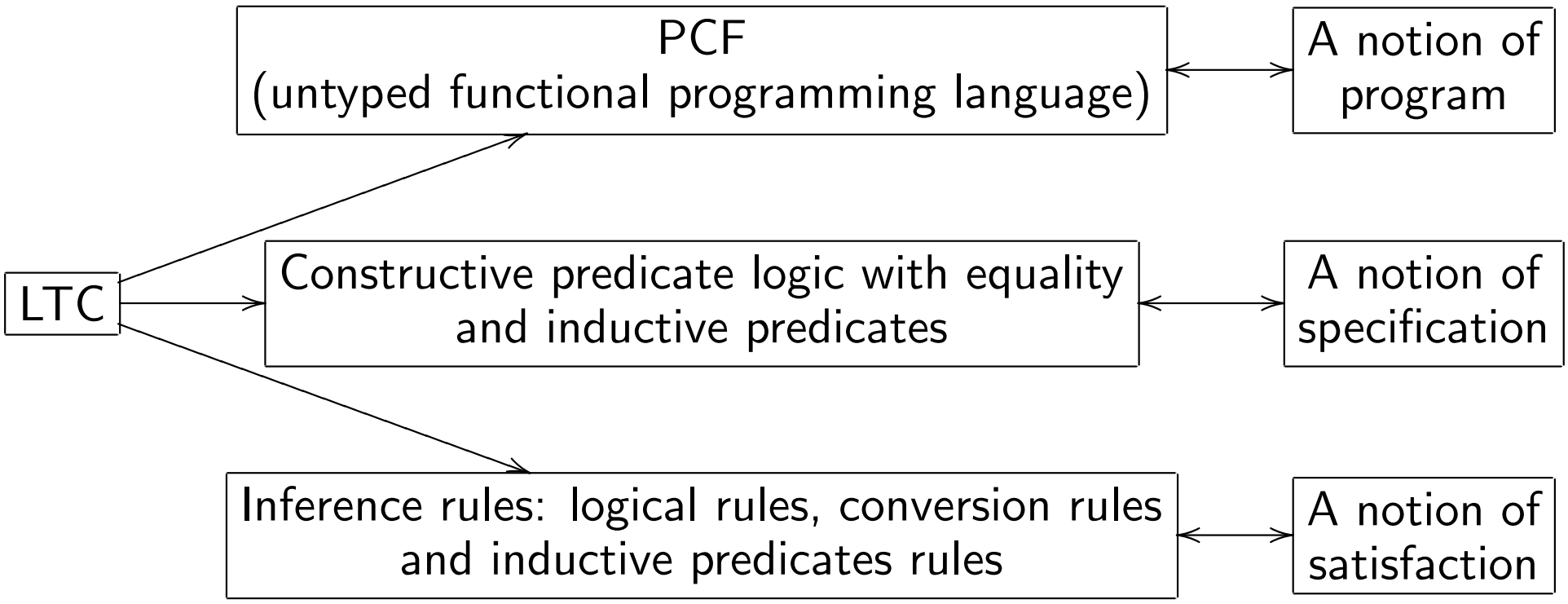
In this talk

1. A core functional programming: Plotkin's PCF
2. A programming logic: Aczel's Logical Theory of Constructions (LTC)
3. A proof assistant: Agda, based on intuitionistic type theory (ITT), development at Chalmers

$\Longrightarrow$ Embedding a logical theory of constructions for PCF in Agda

LTC as a programming logic for PCF

LTC (P. Aczel 1974, 1980 and J. Smith 1978, 1984)
LTC as a programming logic (P. Dybjer 1985, 1986, 1990)



LTC as a programming logic for PCF: programming language

PCF-terms

$$t ::= x \mid t\ t \mid \lambda x.t \mid \text{fix}\ x.t \mid 0 \mid \text{succ}\ t \mid \text{pred}\ t \mid \text{iszero}\ t$$
$$\mid \text{true} \mid \text{false} \mid \text{if}\ t\ \text{then}\ t\ \text{else}\ t \mid \text{error}$$

Example (The greatest common divisor (gcd) of two naturals numbers using Euclid's algorithm)

$$\begin{aligned}
&\text{fix}\ g.\lambda m.\lambda n.\text{if}\ (\text{iszero}\ n)\\
&\qquad \text{then if}\ (\text{iszero}\ m)\ \text{then error else}\ m\\
&\qquad \text{else if}\ (\text{iszero}\ m)\ \text{then}\ n\\
&\qquad\qquad \text{else if}\ m \succ n\ \text{then}\ g\ (m-n)\ n\\
&\qquad\qquad\qquad \text{else}\ g\ m\ (n-m)
\end{aligned}$$

LTC as a programming logic for PCF: inference rules

Logical rules

Inference rules for (intuitionistic) predicate logic

Equality rules

$$t = t \text{ (reflexivity)} \qquad s = t \to P\ s \to P\ t \text{ (substitution)}$$

Conversion rules for the PCF-terms

$$\forall t\ t'.\text{if true then } t \text{ else } t' = t$$
$$\forall t\ t'.\text{if false then } t \text{ else } t' = t'$$
$$\text{pred } 0 = 0$$
$$\forall t.\text{pred (succ } t) = t$$
$$\text{iszero } 0 = \text{true}$$
$$\forall t.\text{iszero (succ } t) = \text{false}$$
$$\forall t\ t'.(\lambda x.t)\ t' = t[x := t']$$
$$\forall t.\ \text{fix } x.\ t = t[x := \text{fix } x.\ t]$$

LTC as a programming logic for PCF: inference rules (cont.)

Discrimination rules for constructors

$$\neg(\mathsf{true} = \mathsf{false}) \qquad \forall t.\ \neg(0 = \mathsf{succ}\, t)$$

Introduction rules for B (total booleans) and N (total natural numbers)

$$\frac{}{\mathsf{B}\ \mathsf{true}} \qquad \frac{}{\mathsf{N}\ 0}$$

$$\frac{}{\mathsf{B}\ \mathsf{false}} \qquad \frac{\forall t.\mathsf{N}\ t \rightarrow \mathsf{N}\ (\mathsf{succ}\ t)}{}$$

Elimination rules for B and N

$$P\ \mathsf{true} \rightarrow P\ \mathsf{false} \rightarrow \forall t.(\mathsf{B}\ t \rightarrow P\ t) \quad \text{(proof by case analysis)}$$

$$P\ 0 \rightarrow \forall t.(\mathsf{N}\ t \rightarrow P\ t \rightarrow P\ (\mathsf{succ}\ t)) \rightarrow \forall t.(\mathsf{N}\ t \rightarrow P\ t) \quad \text{(proof by MI)}$$

LTC as a programming logic for PCF: termination and examples

Using totality predicates for expressing termination

N n: n is a total natural number, that is, a PCF program which returns a total natural number when computed (resp. B b).

Example (The gcd always terminates)

$$\forall m\ n.\mathsf{N}\ m \to \mathsf{N}\ n \to \neg(m = 0 \wedge n = 0) \to \mathsf{N}(\mathsf{gcd}\ m\ n)$$

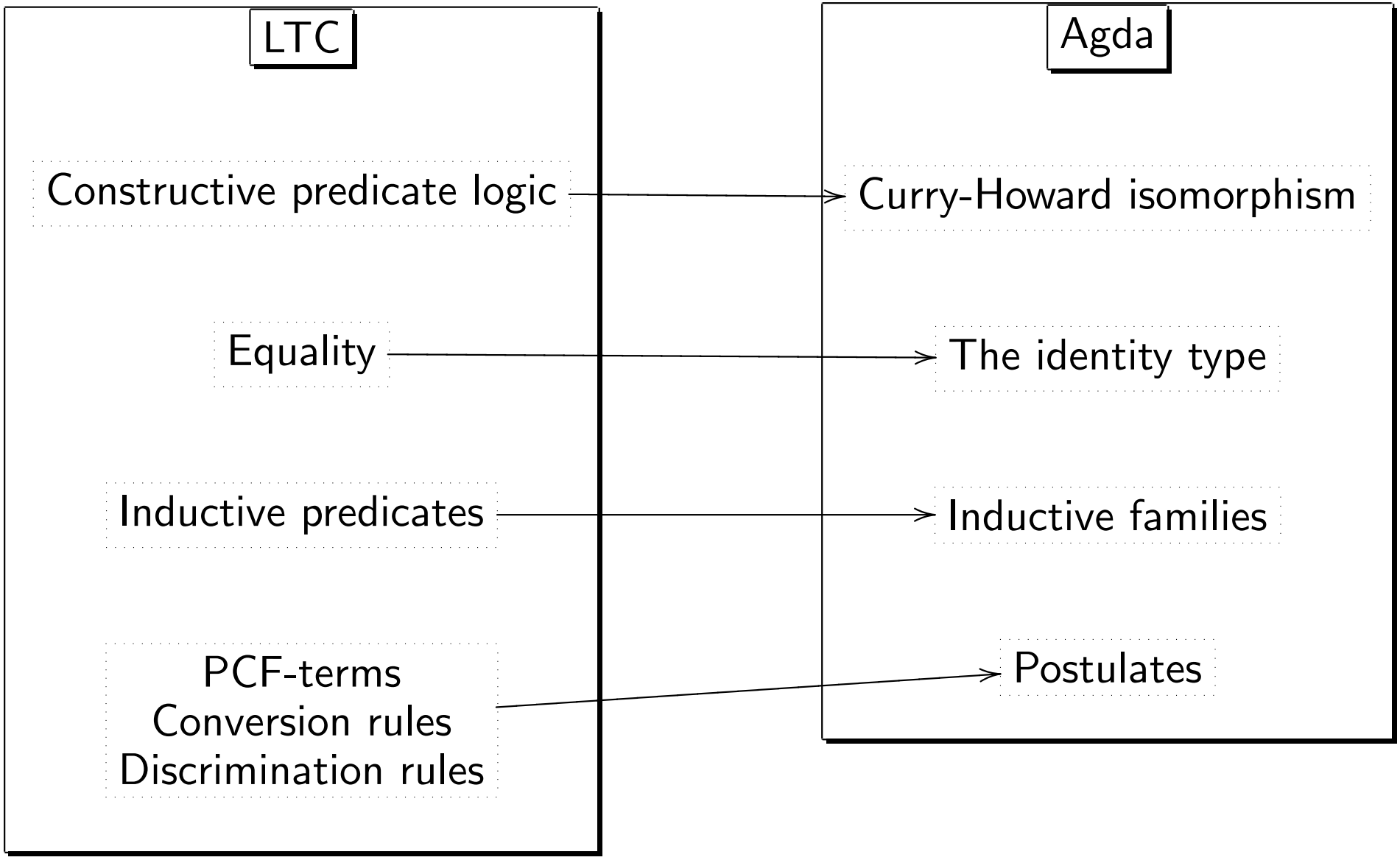
Example (The correctness of the gcd)

$$\begin{aligned}\forall m\ n.\mathsf{N}\ m \to \mathsf{N}\ n \to \neg(m = 0 \wedge n = 0) \to \\ \mathsf{CD}\ m\ n\ (\mathsf{gcd}\ m\ n)\ \wedge \forall d.(\mathsf{CD}\ m\ n\ d \to d \leqslant (\mathsf{gcd}\ m\ n))\end{aligned}$$

where (CD m n d) stands for $(d \mid m \wedge d \mid n)$ when $\_ \mid \_$ is the divisibility predicate.

Agda as a logical framework for LTC

Logical framework-style encoding of LTC: mixed approach



Encoding of LTC

PCF terms (postulates)

```
postulate
  D : Set                          -- The domain of PCF-terms

  λ   : (D -> D) -> D              -- abstraction and app.
  _‘_ : D -> D -> D

  fix : (D -> D) -> D              -- fixed point operator

  zero             : D            -- partial nat. numbers
  succ             : D -> D
  pred, iszero     : D -> D

  true, false    : D               -- partial booleans
  if_then_else_ : D -> D -> D -> D

  error : D                        -- error
```

Encoding of LTC (cont.)

Intuitionistic predicate logic (inductively defined set formers)

```
-- Existential quantification

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

∃-fst : {P : D -> Set} -> ∃ P -> D
∃-fst (∃-i x px) = x

∃-snd : {P : D -> Set} -> (x-px : ∃ P) -> P (∃-fst x-px)
∃-snd (∃-i x px) = px
```

Encoding of LTC (cont.)

The equality predicate (the identity type)

```
data _==_ (x : D) : D -> Set where
  ==-refl : x == x

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

Discrimination rules (postulates)

```
postulate
  true≠false : ¬ (true == false)
  0≠S        : {n : D} -> ¬ (zero == succ n)
```

Encoding of LTC (cont.)

Conversion rules (postulates)

```
postulate

  -- Conversion rules for predecessor
  CP₁ : pred zero                    == zero
  CP₂ : (n : D) -> pred (succ n) == n

  -- The beta-rule
  beta : (f : D -> D) -> (a : D) -> (λ f) ‘ a == f a

  -- Conversion rule for fixed points
  Cfix : (f : D -> D) -> fix f == f (fix f)
```

Encoding of LTC (cont.)

Totality predicates (inductive families)

```
-- Introduction rules for the totality predicate
-- for natural numbers
data N : D -> Set where
  N-z : N zero
  N-s : {n : D} -> N n -> N (succ n)

-- Elimination rule for N
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 N-z      = p0
N-ind P p0 h (N-s Nn) = h Nn (N-ind P p0 h Nn)
```

Example: greatest common divisor

The gcd algorithm

```
gcdh : D -> D
gcdh = \g -> λ (\m -> λ (\n ->
                if (iszero n)
                then (if (iszero m)
                      then error
                      else m)
                else (if (iszero m)
                      then n
                      else (if (m ≻ n)
                            then g ‘ (m - n) ‘ n
                            else g ‘ m ‘ (n - m)))))


gcd : D -> D -> D
gcd m n = fix gcdh ‘ m ‘ n
```

Example: greatest common divisor

Recursive equations

```
gcd-00 : gcd zero zero == error

gcd-S0 : {m : D} -> N m -> gcd (succ m) zero == succ m

gcd-0S : {n : D} -> N n -> gcd zero (succ n) == succ n

gcd-S>S : {m n : D} -> N m -> N n -> (succ m > succ n) ->
          gcd (succ m) (succ n) ==
                      gcd (succ m - succ n) (succ n)

gcd-S≤S : {m n : D} -> N m -> N n -> succ m ≤ succ n ->
          gcd (succ m) (succ n)  ==
                      gcd (succ m) (succ n - succ m)
```

Example: termination of gcd

We want to prove the termination property

```
gcd-N : {m n : D} -> N m -> N n ->
        ¬ ((m == zero) ∧ (n == zero)) ->
        N (gcd m n)
```

Example: termination of gcd (cont.)

Auxiliary lemmas

```
gcd-S0-N : {m : D} -> N m -> N (gcd (succ m) zero)

gcd-0S-N : {n : D} -> N n -> N (gcd zero (succ n))

gcd-S>S-N : {m n : D} -> N m -> N n ->
            N (gcd (succ m - succ n) (succ n)) ->
            succ m > succ n ->
            N (gcd (succ m) (succ n))

gcd-S≤S-N : {m n : D} -> N m -> N n ->
            N (gcd (succ m) (succ n - succ m)) ->
            succ m ≤ succ n ->
            N (gcd (succ m) (succ n))
```

Example: termination of gcd (cont.)

Auxiliary lemma for the case $m > n$ (similar for the case $m \leq n$)

```
gcd-x>y-N :  {m n : D} -> N m -> N n ->
             ... ->
             m > n ->
             ¬ ((m == zero) ∧ (n == zero)) ->
            N (gcd m n)
gcd-x>y-N = -- Using pattern matching on the proofs that
            -- m and n are totals
```

Example: termination of gcd (cont.)

The proof

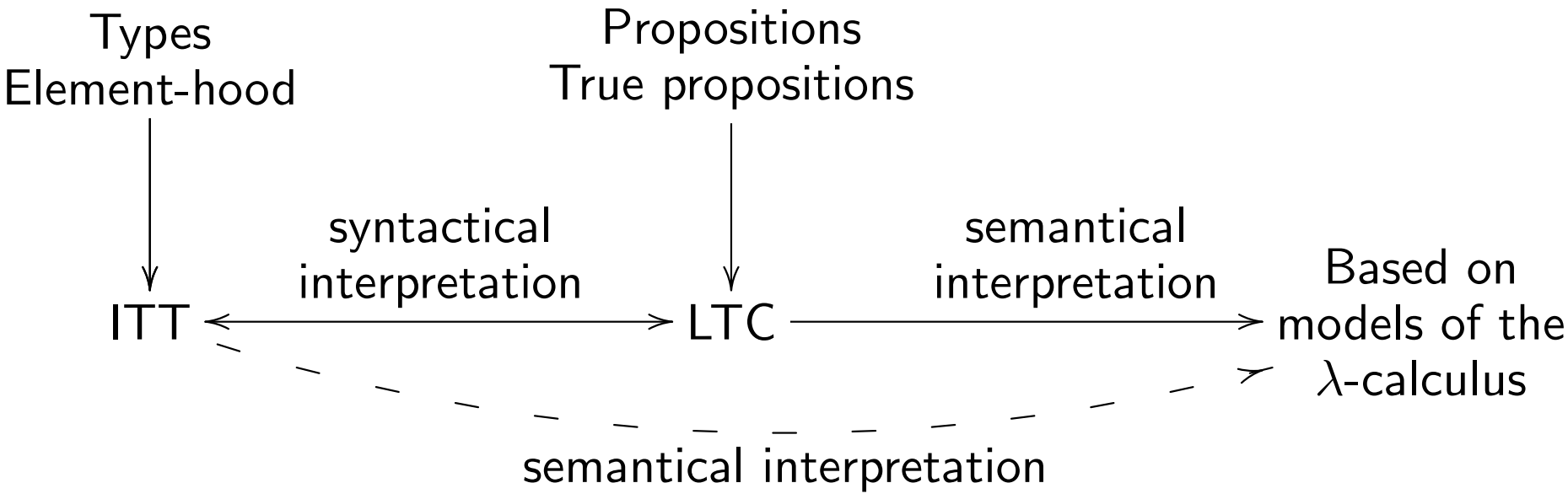
```
gcd-N : {m n : D} -> N m -> N n ->
        ¬ ((m == zero) ∧ (n == zero)) ->
        N (gcd m n)
gcd-N Nm Nn = N-wf₂ P istep Nm Nn
  where
  P : D -> D -> Set
  P i j = ¬ ((i == zero) ∧ (j == zero)) -> N (gcd i j )

  istep :
    {i j : D} -> N i -> N j ->
    ({i' j' : D} -> N i' -> N j' ->
      (i' , j') <₂ (i , j) -> P i' j') ->
    P i j
  istep Ni Nj allAcc = ∨-elim (gcd-x>y-N Ni Nj allAcc )
                              (gcd-x≤y-N Ni Nj allAcc )
                              (x>y∨x≤y Ni Nj)
```

LTC: Original motivation

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

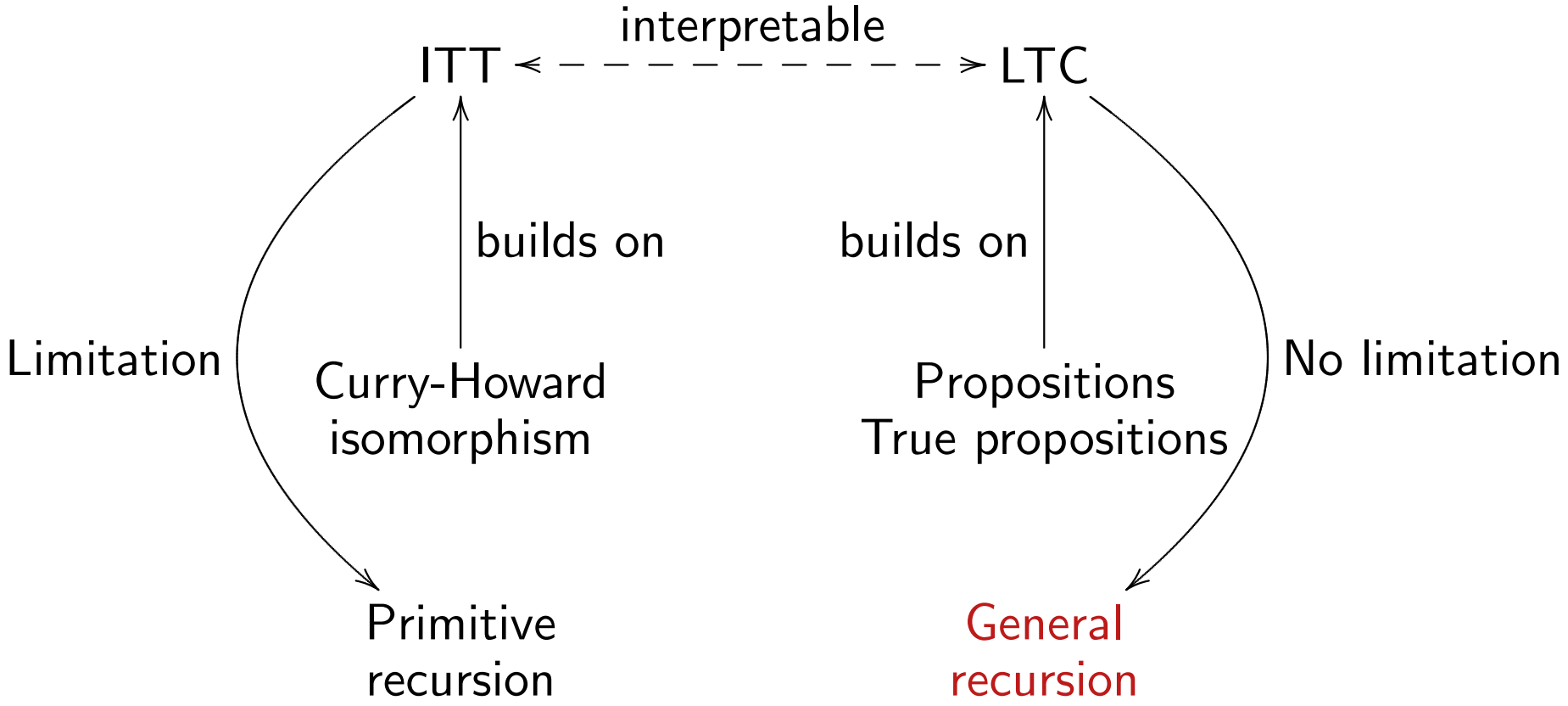
> *"The basic LTC framework is intended to be, at the informal level, the framework of ideas that are being used by Per Martin-Löf in his semantical explanations for ITT. Those explanations seem to treat the notions of proposition and truth as fundamental and use them to explain the notions of type and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

LTC: Original motivation (cont.)



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)

Future work

- Plug-in for automatic theorem prover
- Using Agda's standard library for proofs in LTC
- Comparing LTC with others programming logics and comparing Agda/LTC with others proof-assistants.
- More functional programming language features

Final remarks

LTC is an appropriate constructive programming logic for reasoning about general recursive functional programs:

- It does not have 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 amounts to proving its termination.
- It is at least as strong as ITT.

References I

[Acz77] Peter Aczel.

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

In S. Miettinen and J. Väänanen, editors, *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 Jon Barwise, H. Jerome Keisler, and Kenneth Kunen, editors, *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 31–59. Amsterdan: North-Holland, 1980.

References II

[Dyb85] Peter Dybjer.

Program verification in a logical theory of constructions.

In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 334–349, 1985.

Appears in revised form as Programming Methodology Group Report 26, June 1986.

[Dyb86] Peter Dybjer.

Program verification in a logical theory of constructions.

Technical Report Programming Methodology Group, Report 26, University of Gothenburg 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.

References III

[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.

[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 University of Gothenburg, 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.