Inductive Definitions

Christine Paulin-Mohring
Université Paris Sud, Orsay, France

June 2003

Introduction : what is an inductive definition ?

Informal definition

An inductive definition introduces a **new concept** by :

- specifying a set of **rules of constructions**
- the assumption that we are considering the **smallest notion** closed under the rules of constructions
- different rules of constructions give different objects

Inductive definitions are everywhere !

- programming
  - data structures : records, lists, trees . . .
  - predicate definitions as clauses in logic programming
- semantics of programming languages
  - abstract syntax trees
  - static, operational semantics as inference rules
- logic
  - representation of terms, formulas
  - semantics, deduction relation
  - constructive interpretation of connectors

Two different views

Mathematics

- Make sure the notion is well-defined
- Implicit justification in main cases
- Extensional view of functions, equational reasoning

Programming language or proof assistant

- Explicit construction
- Intensional view, computation
- Reasoning

How to provide inductive definitions ?

in proof-assistant (HOL, type theory)

- Representation
  - primitive notion / encoded
- Which class of inductive definitions ?
  - strictly positive, positive, no restriction
  - polymorphic, dependent types
- Which rules ?
  - primitive rules / derived rules

What we expect from an inductive definition ?

- Termination of computations
- Consistency of the underlying logic
- Representation theorems
  - Correspondance between terms in an inductive type and the values to be represented
- Provability of properties of inductive definitions inside the proof assistant

Contents of the course

1. Different sorts of inductive types
2. Inductive relations
3. Inductive definitions in different systems
4. Applications

Different sorts of inductive types

- Examples of inductive types
  - Algebraic structures, ML concrete types and others.
- Non-recursive concrete types and logic
- Recursive definitions

Algebraic Types

Introduce sorts, constants, operators with signature

Natural numbers `nat`

- **constant** `0`,
- **operator** `S`: (`nat`) $\rightarrow$ `nat`

Finite sets `color`

- **constants** `red`, `black`

Lists of natural numbers `list`

- **constant** `nil`,
- **operator** `cons` (`nat`,`list`) $\rightarrow$ `list`.

Expected properties : Free (initial) algebra

Injectivity $\texttt{Cons}(n_1, l_1) = \texttt{Cons}(n_2, l_2) \Rightarrow n_1 = n_2 \wedge l_1 = l_2$

Non-confusion $\texttt{Nil} \neq \texttt{Cons}(n, l)$

Finite terms any object of sort `list` is finitely built with constant `Nil` and operator `Cons`.

Induction

$$\frac{P(\texttt{Nil}) \qquad \forall n : \texttt{nat}.\forall l : \texttt{list}.P(l) \Rightarrow P(\texttt{Cons}(a, l))}{\forall l : \texttt{list}.P(l)}$$

Associated functions

Testing *isNil*,*isCons* : `list`$\rightarrow$`bool`

$$(\mathit{isNil}\ \texttt{Nil}) \longrightarrow \texttt{true} \qquad (\mathit{isNil}\ \texttt{Cons}(n, l)) \longrightarrow \texttt{false}$$

Projections partial functions on non-nil lists

$hd : \texttt{list}^* \rightarrow \texttt{int}$	$tl : \texttt{list}^* \rightarrow \texttt{list}$
$(hd\ \texttt{Nil}) \longrightarrow \perp$	$(tl\ \texttt{Nil}) \longrightarrow \perp$
$(hd\ \texttt{Cons}(n, l)) \longrightarrow n$	$(tl\ \texttt{Cons}(n, l)) \longrightarrow l$

Inductive types in functional programming languages

OCAML notations

```
type list = Nil | Cons of int * list
type color = Red | Black
```

Parameterized types

```
type 'a option = None | Some of 'a
type 'a list = Nil | Cons of 'a * 'a list
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

Expected Properties

Any value of type `list` evaluates to :

$$\texttt{Nil} \text{ or } \texttt{Cons}(n, l)$$

with n of type `int` and l of type `list`

Non confusion

$$\texttt{Nil} \neq \texttt{Cons}(n, l)$$

Injectivity

$$\texttt{Cons}(n_1, l_1) = \texttt{Cons}(n_2, l_2) \Leftrightarrow n_1 = n_2 \text{ and } l_1 = l_2$$

Pattern-matching

match x with Nil → f | Cons(n,l) → g

Equivalent to testing and projections functions :

let *isNil* x = match x with Nil → *true* | Cons(n,l) → *false*
let *hd* x = match x with Nil → *failwith* "hd" | Cons(n,l) → n
let *tl* x = match x with Nil → *failwith* "tl" | Cons(n,l) → l

Compiling pattern-matching

match x with Nil → f | Cons(n,l) → g

≡

if *isNil* x then f else let n = *hd* x and l = *tl* x in g

Properties

Assume equality is substitutive (Leibniz)

$$\frac{x = y \qquad P(x)}{P(y)} \qquad\qquad \frac{x = y}{f(x) = f(y)}$$

The pattern-matching construction (or test and projection functions) allows to derive :

- Nil=Cons(a,l) ⇒ true=false
- $\texttt{Cons}(a, l) = \texttt{Cons}(b, m) \Leftrightarrow a = b \wedge l = m$

Tree with finite (unbounded) branching

```
type tree = Node of tree list
```

- No explicit constant but still a basic element Node(Nil)
- Recursive argument as a parameter of another inductive type
- Could also be seen as mutual inductive definitions : `tree` and `tree_list`

Non-algebraic inductive types

Trees with infinite (denumerable) branching

```
type itree = Leaf | Node of (nat → itree)
```

$$\frac{}{\texttt{Leaf} : \texttt{itree}} \qquad \frac{f : \texttt{nat} \to \texttt{itree}}{\texttt{Node}(f) : \texttt{itree}}$$

f:`nat` →`itree` is an infinite family of trees indexed by a natural number i.
`Node`(f) is a tree with a root and a denumerable set of subtrees $f_0, f_1, \ldots, f_n, \ldots$ represented by f.

Remark : $\texttt{Node}(f) = \texttt{Node}(g) \Leftrightarrow f = g \not\Leftrightarrow \forall n : \texttt{nat}.(f\ n) = (g\ n)$

Non-positive types

type `lambda` $=$ `Lam` of `lambda` $\to$ `lambda` $\qquad \frac{f : \texttt{lambda} \to \texttt{lambda}}{\texttt{Lam}(f) : \texttt{lambda}}$

$\boxed{\lambda\text{-term representation}}$

`lambda`	$\lambda - \text{term}$
`Lam`(fun $x \to x$)	$\lambda x.x$
`Lam`(fun $x \to$ (`Lam`(fun $y \to x$)))	$\lambda x.\lambda y.x$

Lambda-term representation : application

Application is defined by pattern-matching

app : `lambda` → `lambda` → `lambda`

```
let app (Lam f) x = f x
```

$$\frac{\lambda f.\lambda g.\lambda x.(f\ (g\ x))}{\texttt{Lam}(\text{fun } f \to \texttt{Lam}(\text{fun } g \to \texttt{Lam}(\text{fun } x \to (app\ \ f\ (app\ \ g\ x)))))}$$

Substitution for free

Non Recursive types

$\boxed{\text{Basic constructions}}$

`()`:`unit`
`inl`: $\alpha \to \texttt{sum}_{\alpha,\beta}$, `inr`: $\beta \to \texttt{sum}_{\alpha,\beta}$

$\boxed{\text{Derived notions}}$

`bool`	$\texttt{sum}_{\texttt{unit},\texttt{unit}}$
`true`	`inl(())`
`false`	`inr(())`

Product

Primitive notion in CAML

Can also be seen as an inductive definition
(curryfication of constructor)

$\texttt{pair}$: $\alpha \to \beta \to \texttt{prod}_{\alpha,\beta}$

Any non-recursive concrete data-type in ML can be encoded using $\texttt{prod}$, $\texttt{sum}$ and $\texttt{unit}$

Logic : introduction

Proofs as Programs paradigm

Constructors	Introduction rules
$\texttt{()} : \texttt{unit}$	$\dfrac{}{\top}$
$\texttt{inl} : \alpha \to \texttt{sum}_{\alpha,\beta}$, $\texttt{inr} : \beta \to \texttt{sum}_{\alpha,\beta}$	$\dfrac{A}{A \vee B} \qquad \dfrac{B}{A \vee B}$
$\texttt{pair} : \alpha \to \beta \to \texttt{prod}_{\alpha,\beta}$	$\dfrac{A \quad B}{A \wedge B}$

Other logical constructions : $\bot$ $\exists$
Elimination rules, reduction.

Logic : elimination

pattern-matching/projections	Elimination rules
$\dfrac{x : \texttt{prod}_{\alpha,\beta}}{\mathit{fst}\,(x) : \alpha} \qquad \dfrac{x : \texttt{prod}_{\alpha,\beta}}{\mathit{snd}\,(x) : \beta}$	$\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{B}$
$\dfrac{x : \texttt{sum}_{\alpha,\beta} \quad a : \alpha \vdash f : \gamma \quad b : \beta \vdash g : \gamma}{\mathsf{match}\ x\ \mathsf{with}\ \texttt{inl}(a) \to f \mid \texttt{inr}(b) \to g : \gamma}$	$\dfrac{A \vee B \quad A \vdash C \quad B \vdash C}{C}$
$\dfrac{x : \texttt{prod}_{\alpha,\beta} \quad a : \alpha, b : \beta \vdash p : \gamma}{\mathsf{let}\ (a,b) = x\ \mathsf{in}\ p : \gamma}$	$\dfrac{A \wedge B \quad A, B \vdash C}{C}$
$\mathit{isleft} : \texttt{sum}_{\alpha,\beta} \to \texttt{bool}$ $\mathit{pleft} : \texttt{sum}_{\alpha,\beta} \to A$ $\mathit{pright} : \texttt{sum}_{\alpha,\beta} \to B$	$A \vee B \equiv$ $\exists b : \texttt{bool},$ $b = \texttt{true} \Rightarrow A$ $\wedge\ b = \texttt{false} \Rightarrow B$

Logic : cut elimination

Reductions correspond to cut-elimination :

$$\dfrac{\dfrac{\boxed{A}}{A \vee B} \quad \boxed{A \vdash C} \quad B \vdash C}{C} \quad \longrightarrow \quad \dfrac{\begin{array}{c}\cancel{A}\\ \vdots\end{array}}{C}$$

$$\mathsf{match}\ \texttt{inl}(a)\ \mathsf{with}\ \texttt{inl}(x) \to f \mid \texttt{inr}(y) \to g$$

$$\longrightarrow$$

$$f[x := a]$$

Absurd proposition

$\perp$ corresponds to an empty type

- No introduction rule
- Elimination provides an element in any type

$$\frac{\perp}{C}$$

Recursive types and logic

Specify type `E` with one constructor `C`: `E` $\to$ `E`
There is no finite object in that type.
f : `E` $\to \perp$ can be built for any type $\perp$ (by recursion)

$$f\ (\mathtt{C}\ x) = (f\ x)$$

Specify property `A` with one constructor `neg`: (`A` $\to \perp$) $\to$ `A`
Then $\perp$ is inhabited.
Build a term p of type `A` $\to \perp$,
(`neg` p) has type `A`, (p (`neg` p)) has type $\perp$.

$$(p\ x) \equiv \mathsf{match}\ x\ \mathsf{with}\ \mathtt{neg}\ f \to (f\ x)$$

Dependent types

Types (or propositions) may depend on variables.

- object variable x of type s
- second-order type variables α

Dependent functions have type $\forall x : s.p(x)$, $\forall \alpha.p(\alpha)$.

$$\frac{f : \forall x : s.p(x) \quad t : s}{f\ t : p(t)} \qquad \frac{f : \forall \alpha.p(\alpha)}{f : p(A)}$$

Notational remark In the proposition as type paradigm, one may identify the products

$$\forall x : P.Q \equiv P \Rightarrow Q \qquad \forall \alpha : \mathrm{Prop}.P \equiv \forall \alpha.P$$

Existential quantifiers

$\exists x : s.p(x)$ can also be seen as an inductive definition $\mathtt{sig}_p$

$$\mathtt{dpair} : \forall x : s.p(x) \to \mathtt{sig}_p$$

Dependent pair or disjoint union with constructors indexed by $t : s$

$$\mathtt{dpair}_t : p(t) \to \mathtt{sig}_p$$

Introduction rule/constructor :

$$\frac{t : s \quad u : p(t)}{\mathtt{dpair}\ t\ u : \mathtt{sig}_p} \qquad \frac{p(t)}{\exists x : s.p(x)}$$

Elimination (x is a fresh variable) :

$$\frac{u : \mathtt{sig}_p \quad x : s, y : p(x) \vdash f : C}{\mathsf{let}\ (x, y) = u\ \mathsf{in}\ f : C} \qquad \frac{\exists x : s.p(x) \quad p(x) \vdash C}{C}$$

Second order existential quantification

Second order type $p(\alpha)$ depending on type variable α. $\exists\alpha.p(\alpha)$ can also be seen as an inductive definition $\mathtt{sig}_p$:

$$\mathtt{Eintro} : \forall\alpha.p(\alpha) \to \mathtt{sig}_p$$

Introduction rule/constructor :

$$\frac{u : p(A)}{\mathtt{Eintro}\ A\ u : \mathtt{sig}_p} \qquad \frac{p(A)}{\exists\alpha.p(\alpha)}$$

Elimination (α is a fresh type variable) :

$$\frac{u : \mathtt{sig}_p \quad y : p(\alpha) \vdash f : C}{\mathsf{let}\ (\alpha, y) = u\ \mathsf{in}\ f : C} \qquad \frac{\exists\alpha.p(\alpha) \quad p(\alpha) \vdash C}{C}$$

Constructive interpretation of disjonction

Any value of type $\mathtt{sum}_{\alpha,\beta}$ is either $\mathtt{inl}$(a) or $\mathtt{inr}$(b).
If p is a program of type $\mathtt{sum}_{\alpha,\beta}$ without free variables, then it evaluates to a value of type $\mathtt{sum}_{\alpha,\beta}$.
Logical interpretation : any proof without hypotheses of $A \vee B$ can be reduced by cut-elimination to a proof which ends by

$$\frac{A}{A \vee B} \quad \text{or} \quad \frac{B}{A \vee B}$$

So either A or B are also provable.

Constructive interpretation of existential quantifier

A value in $\mathtt{sig}_p$ is $(\mathtt{dpair}\ t\ u)$ with $u : p(t)$.

Any proof without hypothesis of $\exists x.p(x)$ can be normalized to a proof ending with an introduction rule.

$$\frac{p(t)}{\exists x : s.p(x)}$$

Constructive property :
If $\exists x : s.p(x)$ is provable,
then one can find a term t such that $p(t)$ is provable.

Categorical view

Sum and product

$$\begin{tikzcd} A \arrow[r] \arrow[d] & C \\ A+B \arrow[ur, dashed] & B \arrow[l] \arrow[u] \end{tikzcd} \qquad \begin{tikzcd} A & C \arrow[l] \arrow[d] \arrow[dl, dashed] \\ A*B \arrow[u] \arrow[r] & B \end{tikzcd}$$

Recursion : monotonic functor $X \mapsto F(X)$

$$\begin{tikzcd} \mathsf{c} : F(I) \arrow[r] \arrow[d, dashed, "F(\mathsf{rec} f)"'] & I \arrow[d, dashed, "\mathsf{rec} f"] \\ f : F(C) \arrow[r] & C \end{tikzcd} \qquad \mathsf{rec} f\,(\mathsf{c}\ p) = f\,(F\ \mathsf{rec} f)\ p$$

Dual notion : co-inductive types

Primitive recursion

$g : \mathtt{nat}^k \to \mathtt{nat}$, $h : \mathtt{nat}^{k+2} \to \mathtt{nat}$

define a unique function $f : \mathtt{nat}^{k+1} \to \mathtt{nat}$ s.t.

$$\begin{cases} f(0, x_1, \ldots, x_k) & = g(x_1, \ldots, x_k) \\ f(n+1, x_1, \ldots, x_k) & = h(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{cases}$$

Remark Categorical construction gives a weaker (but equivalent) iterative scheme

$$\begin{cases} f(0, x_1, \ldots, x_k) & = g(x_1, \ldots, x_k) \\ f(n+1, x_1, \ldots, x_k) & = h(f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{cases}$$

Recursive definitions

Algebraic view (rewriting)

length(Nil)=0

length(Cons(n,l))=1+*length*(l)

Testing and projecting

let rec *length* x = if *isNil* x then 0 else 1+*length* (*tl* x)

Pattern-matching

```
let rec length x = match x with
    Nil → 0
  | Cons(n,l) → 1+(length l)
```


Primitive recursive functionals

Generalisation

$g : \alpha$, $h : \mathtt{nat} \to \alpha \to \alpha$

define a unique function $f : \mathtt{nat} \to \alpha$ s.t.

$$\begin{cases} f(0) & = g \\ f(n+1) & = h(n, f(n)) \end{cases}$$

Gödel system T, a combinator

$R_\alpha : \alpha \to (\mathtt{nat} \to \alpha \to \alpha) \to \mathtt{nat} \to \alpha$ s.t.

$$\begin{cases} R_\alpha\ g\ h\ 0 & \longrightarrow g \\ R_\alpha\ g\ h\ (n+1) & \longrightarrow h\ n\ (R_\alpha\ g\ h\ n) \end{cases}$$

Functional definition of *min*

min is a primitive recursive function.

Easy definition as a primitive recursive functional.

$$\begin{array}{ll} \mathit{min} & : \mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat}) \\ \mathit{min}\ 0\ m & = 0 \\ \mathit{min}\ n+1\ m & = \text{if } m = 0 \text{ then } 0 \text{ else } (\mathit{min}\ n\ (m-1)) + 1 \end{array}$$

Ackermann function

$$\begin{cases} ack\,(0,m) & = m+1 \\ ack\,(n+1,0) & = ack\,(n,1) \\ ack\,(n+1,m+1) & = ack\,(n, ack\,(n+1,m)) \end{cases}$$

Primitive recursive functional

$$\begin{array}{ll} ack & : \mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat}) \\ ack\ 0\ m & = m+1 \\ ack\ n+1\ m & = ack\_aux\ m\ n\ (ack\ n) \end{array}$$

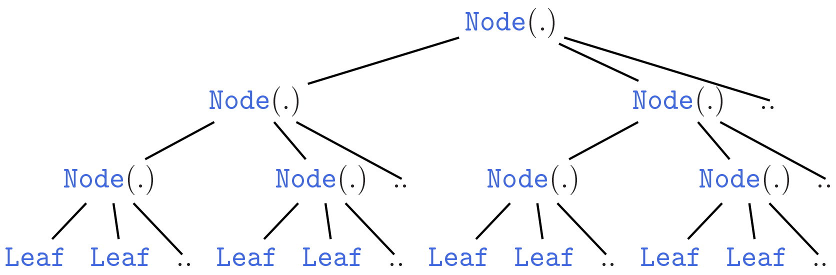
$$\begin{array}{ll} ack\_aux & : \mathtt{nat} \to \mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \\ ack\_aux\ 0\ n\ f & = (f\ 1) \\ ack\_aux\ (m+1)\ n\ f & = (f\ (ack\_aux\ m\ n\ f)) \end{array}$$

Uniform tree of height n

$$uniform : \mathtt{nat} \to \mathtt{itree}$$

let rec *uniform* x = match x with
  0 → Leaf | S(n) → Node(fun p → *uniform* n)

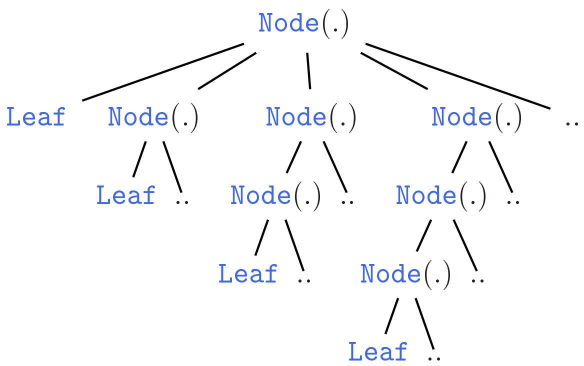
uniform 3 ≡



Tree of infinite height

let *inft* = Node(*uniform*)

inft ≡



Unbounded height

Tree of infinite height (2)

Each branch is finite

Length of a path : $[p_0; p_1; ..; p_n; ..]$

Path represented as a function $p : \mathtt{nat} \to \mathtt{nat}$

lgpath : (nat → nat) → itree → nat
let rec *lgpath* p x = match x with
  Leaf → 0
  | Node(f) → 1+(*lgpath* (*shift* p) (f (p 0)))

$$\forall p.(lgpath\ p\ (uniform\ n)) = n$$
$$\forall p.(lgpath\ p\ inft) = 1 + (p\ 0)$$

Non terminating computations

let *gcd* a b = if a = b then a
else if a < b then *gcd* a (b-a)
else *gcd* (a-b) b

(*gcd* a b) loops when $b \leq 0$ or $a \leq 0$

Unbounded search p :nat→ bool

let *search* n = if p n then n else *search* (n+1)

(*search* n) loops unless $\exists k \geq n.(p\ \ k) = \texttt{true}$

Strong normalisation is lost !

Other recursive schemes

let *gcd* n a b = if n=0 then fail
else if a = b then a
else if a < b then *gcd* (n-1) a (b-a)
else *gcd* (n-1) (a-b) b

Structural recursion
Computes *gcd* with the extra assumption that $a + b < n$
Assuming $a + b < n$,
show $a + (b - a) < n - 1$ and $(a - b) + b < n - 1$
requires $a > 0$ and $a > 0$

Bounded search

Assume p :nat→ bool

let *search* k n = if k=0 then fail
else if p n then n else *search* (k-1) (n+1)

(*search* k n) correct when $\exists l.n \leq l < n + k.(p\ \ l) = \texttt{true}$

Extra argument protects unwanted reduction and bounds the number of reductions.
Additional cost.

Theoretical result

Any recursive function, provably total in higher-order arithmetic can be represented using (higher-order) primitive recursive scheme.

- Take a recursive function f
- Kleene T,U (primitive recursive) : natural number n which represents f : $f(x) = y \Leftrightarrow \exists k.T(n, x, k) \wedge U(k) = y$
- f is provably total if there is a proof of : $\forall x.\exists k.T(n, x, k)$
- From this proof, one can extract a (higher-order) primitive recursive representation.

Maybe not the appropriate algorithm !

Recursive definition of (infinite) values

```
let rec one = Cons(1,one)
```

$$(\mathit{length\ one}) \longrightarrow 1 + (\mathit{length\ one}) \longrightarrow \cdots$$

Non recursive but non terminating term

```
type lambda = Lam of lambda → lambda
```

```
let delta = Lam (fun x → app x x)
let omega = app delta delta
```

Reduction

$$\begin{array}{lll} \mathit{omega} & \equiv & \mathit{app}\ (\mathsf{Lam}\ (\mathsf{fun}\ x \rightarrow \mathit{app}\ x\ x))\ \mathit{delta} \\ & \longrightarrow & (\mathsf{fun}\ x \rightarrow \mathit{app}\ x\ x)\ \mathit{delta} \\ & \longrightarrow & \mathit{app}\ \mathit{delta}\ \mathit{delta} \\ & \equiv & \mathit{omega} \end{array}$$

Recursion

- Limited scheme for structural recursion
  - covers pattern-matching and fixpoint
  - logical point of view
  - classical proofs of normalisation
- Fixpoint
  - requires additional pattern-matching
  - computational point of view
  - criteria for termination

Induction principles

Let $\mathtt{I}$ be an inductive definition with rules of constructions $c_1,\ldots,c_n$

How do I prove $\forall x : \mathtt{I}.P(x)$?

$\boxed{\text{Case analysis}}$

Any value of type $\mathtt{I}$ can be reduced to $c_i(a_1,\ldots,a_k)$

$$\frac{(\forall a_1,\ldots a_k.x = c_i(a_1,\ldots,a_k) \Rightarrow P(x)) \qquad i \in \{1\ldots n\}}{\forall x : \mathtt{I}.P(x)}$$

$$\frac{(\forall a_1,\ldots a_k.P(c_i(a_1,\ldots,a_k))) \qquad i \in \{1\ldots n\}}{\forall x : \mathtt{I}.P(x)}$$

Dependent type for match

$$\frac{x : \mathtt{sum}_{\alpha,\beta} \quad a:\alpha \vdash f : P(\mathtt{inl}(a)) \quad b:\beta \vdash g : P(\mathtt{inr}(b))}{\mathsf{match}\ x\ \mathsf{with}\ \mathtt{inl}(a) \to f | \mathtt{inr}(b) \to g : P(x)}$$

$$\frac{x : \mathtt{prod}_{\alpha,\beta} \quad a:\alpha, b:\beta \vdash p : P(\mathtt{pair}\ a\ b)}{\mathsf{let}\ (a,b) = x\ \mathsf{in}\ p : P(x)}$$

$\boxed{\text{General case}}$

$$\frac{x : \mathtt{I} \qquad (a_1, \ldots a_k \vdash f_i : P(c_i(a_1,\ldots,a_k))) \qquad i \in \{1 \ldots n\}}{\mathsf{match}\ x\ \mathsf{with}\ \ldots | c_i(a_1,\ldots,a_k) \to f_i | \ldots : P(x)}$$

Properties

Types are preserved by reduction

$$\begin{array}{ll} \mathsf{match}\ \mathtt{inl}(t)\ \mathsf{with}\ \mathtt{inl}(a) \to f | \mathtt{inr}(b) \to g & : P(\mathtt{inl}(t)) \\ \downarrow & \\ f[a := t] & : P(\mathtt{inl}(t)) \end{array}$$

Consequence on disjonction

Assume

classic : ∀ A. A ∨ ¬ A

diff : `true` ≠ `false`

Define

witness A = match *classic* A with
    inl a → true | inr na → false

Prove $\forall A.(\mathit{witness}\ A) = \mathtt{true} \Leftrightarrow A$

by case analysis on the proof of $(\mathit{classic}\ A)$

Building a paradox

`bool : Prop`

witness : `Prop → bool`

T : `bool → Prop := [b]b=true`

∀ A. (*T* (*witness* A)) ⇔ A)

leads to a paradox (Girard) : J. Hurkens adapted by H. Geuvers
look at `coq/theories/Logic/Hurkens.v`

Deduce $(\forall A.A \vee \neg A) \Rightarrow \mathtt{true} = \mathtt{false}$

Classical logic implies proof-irrelevance

Consequences on existential type

$\mathtt{sig}_p$ admits two projections

$$\begin{aligned} \pi_1 &: \mathtt{sig}_p \to s \\ \pi_1\ t &= \mathsf{let}\ (x,y) = t\ \mathsf{in}\ x \end{aligned}$$

Dependent case analysis :

$$\frac{u : \mathtt{sig}_p \quad x : s, y : p(x) \vdash f : P(\mathtt{dpair}\ x\ y)}{\mathsf{let}\ (x,y) = u\ \mathsf{in}\ f : P(u)}$$

Defines $P(u) \equiv p(\pi_1(u))$

$P(\mathtt{dpair}\ x\ y)$ is equivalent (up to computation) to $p(x)$.

$$\frac{u : \mathtt{sig}_p \quad x : s, y : p(x) \vdash y : P(\mathtt{dpair}\ x\ y)}{\mathsf{let}\ (x,y) = u\ \mathsf{in}\ y : P(\pi_1\ u)}$$

Induction

Captures the fact that we consider the smallest set of objects built with constructors.

Consider only strictly positive inductive definitions $\mathtt{I}$.

If a rule of construction has type $A_1 \to \cdots \to A_k \to \mathtt{I}$ then

- either $\mathtt{I}$ doest not occur in A_i
- or $A_i \equiv C_1 \to \cdots \to C_l \to \mathtt{I}$ and $\mathtt{I}$ doest not occur in C_j

$$\frac{(\forall a_1, \ldots a_k. \cdots (\forall y_1 \ldots y_l.(P\ (a_i\ y_1, \ldots y_l))) \to \cdots (P\ (c_i(a_1, \ldots, a_k)))_{i\in\{1\ldots n\}}}{\forall x : \mathtt{I}.P(x)}$$

Induction with more general definitions

Dependent types :

Rule of construction has type : $\forall a_1 : A_1 \ldots \forall a_k : A_k.\mathtt{I}$

Recursive argument : $A_i \equiv \forall y_1 : C_1 \ldots \forall y_l : C_l.\mathtt{I}$

Same rule applies.

Multiple inductive definition : $\mathtt{Node} : \mathtt{tree\ list} \to \mathtt{tree}$

$$\frac{\forall f : \mathtt{tree\ list}.\mathit{all\_list}\,(P, f) \to P(\mathtt{Node}(f))}{\forall x : \mathtt{tree}.P(x)}$$

$$\frac{}{\mathit{all\_list}\,(P, \mathtt{Nil})} \qquad \frac{P(a) \quad \mathit{all\_list}\,(P, l)}{\mathit{all\_list}\,(P, \mathtt{Cons}(a, l))}$$

Exemple : infinitely branching trees

$$\frac{P(\mathtt{Leaf}) \quad \forall f : \mathtt{nat} \to \mathtt{itree}.(\forall n : \mathtt{nat}.P(f\ n)) \Rightarrow P(\mathtt{Node}(f))}{\forall x : \mathtt{itree}.P(x)}$$

We can associate to this rule, an operator for structural recursion on the type.

$$\frac{b : P(\mathtt{Leaf}) \quad r : \forall f : \mathtt{nat} \to \mathtt{itree}.(\forall n : \mathtt{nat}.P(f\ n)) \Rightarrow P(\mathtt{Node}(f))}{\mathit{itree\_rec} : \forall x : \mathtt{itree}.P(x)}$$

$$\text{with } \begin{cases} \mathit{itree\_rec}\,(\mathtt{Leaf}) & = b \\ \mathit{itree\_rec}\,(\mathtt{Node}(f)) & = r\ f\ (\lambda\ n.\mathit{itree\_rec}\,(f\ n)) \end{cases}$$

Basic rules of Martin-Löf's Type Theory/ Strong normalisation.

Definition of types/properties by induction

We often define a type or property by induction on a term.

$$\begin{array}{ll} P(0) & \Leftrightarrow G \\ P(n+1) & \Leftrightarrow H(n, P(n)) \end{array}$$

Examples

$$\mathtt{arr}(0) \Leftrightarrow A \qquad \mathtt{arr}(n+1) \Leftrightarrow A \to \mathtt{arr}(n)$$

$$\mathtt{even}(0) \Leftrightarrow \top \qquad \mathtt{even}(n+1) \Leftrightarrow \neg\mathtt{even}(n)$$

Definition of properties by induction

Non monotonic constructions

Kripke semantics : $W \vdash P$

$$W \vdash (P \Rightarrow Q) \Leftrightarrow \forall W'.W \subseteq W' \Rightarrow (W' \vdash P) \Rightarrow (W' \vdash Q)$$

Reducibility candidate : $t \in [T]_\rho$

$$\begin{array}{ll} t \in [P \Rightarrow Q]_\rho & \Leftrightarrow \forall x.x \in [P]_\rho \Rightarrow (t\ x) \in [Q]_\rho \\ t \in [\forall \alpha.P]_\rho & \Leftrightarrow \forall C.t \in [P]_{\rho,\alpha \mapsto C} \end{array}$$

More with inductive relations

Strong elimination scheme

Replace equivalence by definitional equality

Proposition definition	Proof/object definition
$P(0) = G$	$f(0) = g$
$P(n+1) = H(n, P(n))$	$f(n+1) = h(n, P(n))$

A general scheme

$$\frac{T(n)\ \mathtt{type} \qquad g : T(0) \qquad h : \forall x.T(x) \Rightarrow T(x+1)}{f : \forall x.T(x)}$$

with f the unique function such that $\begin{cases} f(0) & = g \\ f(n+1) & = h(n, f(n)) \end{cases}$

Strong elimination : consequences

- Computation on propositions/types
- Strong elimination for `bool` gives a proof of $\mathtt{true} \neq \mathtt{false}$ (otherwise, proof-irrelevance model)

$$\begin{cases} T & : \mathtt{bool} \to \mathrm{Prop} \\ T(\mathtt{true}) & = \top \\ T(\mathtt{false}) & = \bot \end{cases} \qquad \dfrac{\dfrac{\dfrac{\mathtt{true} = \mathtt{false}}{T(\mathtt{true}) = T(\mathtt{false})}}{\top = \bot \qquad () : \top}}{\bot}$$

Strong elimination and second-order existential

$$\frac{C\ \mathsf{type} \qquad u : \mathtt{sig}_p \qquad y : p(\alpha) \vdash f : C}{\mathsf{let}\ (\alpha, y) = u\ \mathsf{in}\ f : C}$$

Take $p(\alpha) = \top$ $(\exists \alpha.\top)$ and $C = \mathrm{Prop}$

Define $B = \mathtt{sig}_p : \mathrm{Prop}$

Define $\pi : B \to \mathrm{Prop} = [u]\mathsf{let}\ (\alpha, y) = u\ \mathsf{in}\ \alpha$

Define $\epsilon : \mathrm{Prop} \to B = [\alpha]\mathtt{Eintro}(\alpha, ())$

$$\pi(\epsilon\alpha) = \alpha$$

Leads to a paradox!

Strong elimination is only acceptable for small types
(arguments of constructors are specific propositions or sets not the universe of all propositions or sets)

Strong elimination and programming

Non ML-typable program

$$\frac{b : \mathtt{bool} \qquad 0 : T(\mathtt{true}) \qquad (0,0) : T(\mathtt{false})}{\mathsf{match}\ b\ \mathsf{with}\ \mathtt{true} \to 0 \mid \mathtt{false} \to (0,0) : T(b)}$$

$$\begin{cases} T & : \mathtt{bool} \to \mathtt{type} \\ T(\mathtt{true}) & = \mathtt{nat} \\ T(\mathtt{false}) & = \mathtt{nat} * \mathtt{nat} \end{cases}$$

No value in $T(b)$ when b does not start with a constructor.
First eliminate b!

Inductive types : summary

- Algebraic types, ML data types
- Dependent pair, second-order existential
- Non positive inductive definitions have unwanted behavior
- Strong relation between computation and deduction
- Primitive recursive definitions
- Other recursive definitions may be problematic

More on that, when studying properties

Inductive relations

- Logic programming
- Semantics
- Equality
- Well-founded relations
- Recursive definitions of types
- Dependent types

Logic programming

$\texttt{inf}(a,l)$ when a is less than any element of the list l

```
inf(a,Nil) :-
inf(a,Cons(b,l)) :- a ≤ b, inf(a,l)
```

$\texttt{sorted}(l)$ when l is a sorted list

```
sorted(Nil) :-
sorted(Cons(a,l)) :- inf(a,l),sorted(l)
```

$\texttt{insert}(a,l,m)$ when the list m is obtained by insertion of a in l

```
insert(a,Nil,Cons(a,Nil)) :-
insert(a,Cons(b,l),Cons(a,Cons(b,l))) :- a ≤ b
insert(a,Cons(b,l),Cons(b,m)) :- b < a,insert(a,l,m)
```


Expected properties

- Well-defined notion
- Executable specification
  $\texttt{insert}(2,\texttt{Cons}(1,\texttt{Cons}(3,\texttt{Nil})),?)$
  Find m such that $\texttt{insert}(2,\texttt{Cons}(1,\texttt{Cons}(3,\texttt{Nil})),m)$ holds
- Establish properties
  $\texttt{sorted}(l) \Rightarrow (\texttt{insert}(a,l,m) \Rightarrow \texttt{sorted}(m))$

Inductive definitions in logical systems

Inference rules for minimal logic

$$\frac{P \in \Gamma}{\Gamma \vdash P} \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \qquad \frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$$

Reflexive-transitive closure of a property

$$\frac{}{R^*(x,x)} \qquad \frac{R(x,y)}{R^*(x,y)} \qquad \frac{R^*(x,y) \quad R^*(y,z)}{R^*(x,z)}$$

More operational definition (one proof for one path)

$$\frac{}{R^*(x,x)} \qquad \frac{R^*(x,y) \quad R(y,z)}{R^*(x,z)}$$

Operational semantics

Expressions :

$$e := x | e.f | \ldots$$

Static typing $\gamma \vdash e : \tau$

$$\frac{x : \tau \in \gamma}{\gamma \vdash x : \tau} \qquad \frac{\gamma \vdash e : \texttt{Class}(C) \qquad \texttt{isfield}(f, C, \tau)}{\gamma \vdash e.f : \tau}$$

Operational semantics $m, e \mapsto m', v$

$$\frac{\mathit{access}(m,x,v)}{m, x \mapsto m, v} \qquad \frac{m, e \mapsto m', a \qquad \mathit{access\_field}(m,a,f,v)}{m, e.f \mapsto m', v}$$

General scheme of inductive relations

- Precise the arity of the definition $\mathtt{I}(x_1, \ldots, x_n)$
- Gives rules of constructions :

$$\frac{A_1 \;\ldots\; A_n}{\mathtt{I}(t_1, \ldots, t_n)}$$

Implicit quantification over free variables :

$$\forall x_1, \ldots, x_k . A_1 \Rightarrow \cdots A_n \Rightarrow \mathtt{I}(t_1, \ldots, t_n)$$

A_i may contain recursive occurrences of $\mathtt{I}$.

Rules

In case of positive occurrences of $\mathtt{I}$ in A_i.

Introduction rules associated to each rule of construction :

$$\forall x_1, \ldots, x_k . A_1 \Rightarrow \cdots A_n \Rightarrow \mathtt{I}(t_1, \ldots, t_n)$$

Minimality property :
let $P(z_1, \ldots, z_n)$ be a property of same arity

$$\frac{(\forall x_1, \ldots, x_k . A_1[\mathtt{I} := P] \Rightarrow \cdots A_n[\mathtt{I} := P] \Rightarrow P(t_1, \ldots, t_n)) \text{ for all rules}}{\forall z_1, \ldots, z_n . \mathtt{I}(z_1, \ldots, z_n) \Rightarrow P(z_1, \ldots, z_n)}$$

Generalized scheme

Dependent functions in introduction rules

$$\forall x_1 : A_1, \ldots, x_k : A_k . \mathtt{I}(t_1, \ldots, t_n)$$

Recursion instead of iteration scheme

$$\frac{(\forall \vec{x} : \vec{A} . A_{i_1}[\mathtt{I} := P] \Rightarrow \cdots A_{i_l}[\mathtt{I} := P] \Rightarrow P(t_1, \ldots, t_n)) \text{ for all rules}}{\forall z_1, \ldots, z_n . \mathtt{I}(z_1, \ldots, z_n) \Rightarrow P(z_1, \ldots, z_n)}$$

Example : transitive closure

Iteration, first definition

$$\frac{\forall x . P(x, x) \quad \forall x, y . R(x, y) \Rightarrow P(x, y) \quad \forall x, y, z . P(x, y) \Rightarrow P(y, z) \Rightarrow P(x, z)}{\forall x, y . R^*(x, y) \Rightarrow P(x, y)}$$

Iteration, second definition

$$\frac{\forall x . P(x, x) \quad \forall x, y, z . P(x, y) \Rightarrow R(y, z) \Rightarrow P(x, z)}{\forall x, y . R^*(x, y) \Rightarrow P(x, y)}$$

Example : transitive closure, other eliminations

Generalized elimination $Q(x,y) = R^*(x,y) \wedge P(x,y)$

$$\frac{\forall x.P(x,x) \quad \forall x,y,z.R^*(x,y) \Rightarrow P(x,y) \Rightarrow R(y,z) \Rightarrow P(x,z)}{\forall x,y.R^*(x,y) \Rightarrow P(x,y)}$$

Case analysis

$$\frac{\forall x.P(x,x) \quad \forall x,y,z.R^*(x,y) \Rightarrow R(y,z) \Rightarrow P(x,z)}{\forall x,y.R^*(x,y) \Rightarrow P(x,y)}$$

Parameterized version (fix x the basic element)

$$\frac{P(x) \quad \forall y,z.R^*(x,y) \Rightarrow P(y) \Rightarrow R(y,z) \Rightarrow P(z)}{\forall y.R^*(x,y) \Rightarrow P(y)}$$

$Q(x,y) = P(x) \Rightarrow (\forall y,z.R^*(x,y) \Rightarrow P(y) \Rightarrow R(y,z) \Rightarrow P(z)) \Rightarrow P(y)$

Equality

Smallest reflexive binary relation $x = y$

Rule for construction : $\forall x.x = x$

Minimality :

$$\frac{\forall x.P(x,x)}{\forall x,y.x = y \Rightarrow P(x,y)}$$

Fix t, define a unary predicate $t = y$ of terms equal to t ($\mathtt{eq}_t(y)$)

Rule for construction : $t = t$ ($\mathtt{eq}_t(t)$)

Minimality (Leibniz rule)

$$\frac{P(t)}{\forall y.t = y \Rightarrow P(y)} \qquad \frac{t = u \quad P(t)}{P(u)}$$

Equality as an essential notion

The rule of constructor :

$$\forall x_1 : A_1, \ldots, x_k : A_k.\mathtt{I}(t_1, \ldots, t_n)$$

Can be reformulated as :

$$\forall x_1 : A_1, \ldots, x_k : A_k.x = (t_1, \ldots, t_n) \Rightarrow \mathtt{I}(x)$$
$$(\exists x_1 : A_1, \ldots, x_k : A_k.x = (t_1, \ldots, t_n)) \Rightarrow \mathtt{I}(x)$$

Fixpoint

$$\mathtt{I}(x) \Leftrightarrow \bigvee(\exists x_1 : A_1, \ldots, x_k : A_k.x = (t_1, \ldots, t_n))$$

Inductive relations = equality + smallest fixpoint

Example : transitive closure

$$\begin{aligned} R^*(p) \Leftrightarrow \quad & \exists x : A.p = (x,x) \\ & \vee\ \exists x,y,z : A.R^*(x,y) \wedge R(y,z) \wedge p = (x,z) \end{aligned}$$

Alternative caracterisation

$$\begin{aligned} R^*(x,t) \Leftrightarrow \quad & t = x \\ & \vee\ \exists y,z : A.R^*(x,y) \wedge R(y,z) \wedge t = z \end{aligned}$$

Well-founded relation

A binary relation R of type A is well-founded if there is no infinite decreasing sequence : $(f : \mathtt{nat} \to A)(\forall n.R(f(n+1), f(n))) \to \bot$
Constructive definition for an element x of A, to be well-founded.

$$\frac{\forall y.R(y,x) \Rightarrow \mathtt{wf}(y)}{\mathtt{wf}(x)}$$

Branching on x parameterized by $\{y : A | R(y,x)\}$
Associated minimality (generalized elimination) principle

$$\frac{\forall x.(\forall y.R(y,x) \Rightarrow \mathtt{wf}(y)) \Rightarrow (\forall y.R(y,x) \Rightarrow P(y)) \Rightarrow P(x)}{\forall x.\mathtt{wf}(x) \Rightarrow P(x)}$$

Associating terms to proofs

Follows the same pattern as for inductive types:

- Give a name to each rule of construction, seen as a constructor
- Associate a combinator to the elimination scheme

Example

$\mathtt{R*refl} : \forall x.R^*(x,x)$
$\mathtt{R*add} : \forall x,y,z.R^*(x,y) \Rightarrow R(y,z) \Rightarrow R^*(x,z)$

$$\frac{f : \forall x.P(x,x) \qquad g : \forall x,y,z.R^*(x,y) \Rightarrow P(x,y) \Rightarrow R(y,z) \Rightarrow P(x,z)}{\mathit{R{*}ind}_{f,g} : \forall x,y.R^*(x,y) \Rightarrow P(x,y)}$$

$$\begin{cases} (\mathit{R{*}ind}_{f,g}\ x\ x\ \mathtt{R*refl}(x)) & \longrightarrow f \\ (\mathit{R{*}ind}_{f,g}\ x\ z\ \mathtt{R*add}(x,y,z,p,r)) & \longrightarrow (g\ x\ y\ z\ p\ (\mathit{R{*}ind}_{f,g}\ x\ y\ p)\ r) \end{cases}$$

Well-founded induction

The relation is well-founded if all the elements are well-founded

$\mathit{wfR} : \forall x.\mathtt{wf}(x) \qquad \dfrac{\forall x.(\forall y.R(y,x) \Rightarrow P(y)) \Rightarrow P(x)}{\forall x.P(x)}$

Associated proof-terms

Introduction rule : $\mathtt{wf\_intro} : \forall x.(\forall y.R(y,x) \Rightarrow \mathtt{wf}(y)) \Rightarrow \mathtt{wf}(x)$
Elimination rule :

$$\frac{F : \forall x.(\forall y.R(y,x) \Rightarrow \mathtt{wf}(y)) \Rightarrow (\forall y.R(y,x) \Rightarrow P(y)) \Rightarrow P(x)}{\mathit{wfind}_F : \forall x.\mathtt{wf}(x) \Rightarrow P(x)}$$

$(\mathit{wfind}_F\ x\ (\mathtt{wf\_intro}\ x\ f)) \longrightarrow (F\ x\ f\ \lambda y.\lambda r.(\mathit{wfind}_F\ y\ (f\ y\ r)))$

Well-founded recursion

Associated proof-term :

$$\frac{F : \forall x.(\forall y.R(y,x) \Rightarrow P(y)) \Rightarrow P(x)}{(\mathit{fix}\ F) = \lambda x.(\mathit{wfind}_{\lambda x,a,r.(F\ x\ r)}\ x\ (\mathit{wfR}\ x)) : \forall x.P(x)}$$

Well-founded recursion corresponds to structural recursion on a proof of $\mathtt{wf}$(x).
The fixpoint reduction is guarded by the proof $(\mathit{wfR}\ x)$ of $(\mathtt{wf}\ x)$ which should start with a constructor.
When $(\mathit{wfR}\ x) = (\mathtt{wf\_intro}\ x\ f)$,

$(\mathit{fix}\ F\ x) \simeq (F\ x\ \lambda y.\lambda r.(\mathit{fix}\ F\ y))$

Inversion scheme

Elimination schemes proves universally quantified properties

$$\forall x_1,\ldots,x_l.\mathtt{I}(x_1,\ldots,x_l) \Rightarrow P(x_1,\ldots,x_l)$$

No quantification over parameters

- What to do with $\mathtt{I}(t_1,\ldots,t_l) \Rightarrow P(t_1,\ldots,t_l)$ when t_i are not variables ?
- Generalize to $\forall x_1,\ldots,x_l.\mathtt{I}(x_1,\ldots,x_l) \Rightarrow P(x_1,\ldots,x_l)$
- What to do if the generalization does not hold ?
- Find another one which works :
  $\forall x_1,\ldots,x_l.\mathtt{I}(x_1,\ldots,x_l) \Rightarrow (x_1,\ldots,x_l) = (t_1,\ldots,t_l) \Rightarrow P(t_1,\ldots,t_l)$
  Use non-recursive elimination on $\mathtt{I}$

Inversion : example $\mathtt{Le}(n,m)$

Constructors :

$\mathtt{Le0} : \forall m.\mathtt{Le}(0,m)$ $\quad$ $\mathtt{LeS} : \forall n,m.\mathtt{Le}(n,m) \Rightarrow \mathtt{Le}(n+1,m+1)$

Elimination :

$$\frac{\forall m.P(0,m) \quad \forall n,m.\mathtt{Le}(n,m) \Rightarrow P(n,m) \Rightarrow P(n+1,m+1)}{\forall n,m.\mathtt{Le}(n,m) \Rightarrow P(n,m)}$$

How to prove $\forall p,q.\mathtt{Le}(p+1,q+1) \Rightarrow \mathtt{Le}(p,q)$?

Take $P(n,m) = n = p+1 \Rightarrow m = q+1 \Rightarrow \mathtt{Le}(p,q)$

Each case either eliminated by contradiction $0 \neq q+1$
or simplified $n+1 = p+1 \Rightarrow n = p$

Exercise : Prove $\forall k.\mathtt{Le}(k+1,0) \Rightarrow \bot$?

Proofs as objects

Consider proofs of inductive relations as objects

- Size of a derivation
- Path associated to the transitive closure

Define recursive functions :

$\mathit{path} : \forall x,y : A.(R^*\ x\ y) \to (\mathtt{list}\ A)$
$\mathit{path}\ x\ x\ \mathtt{R*refl}(x) = \mathtt{Nil}$
$\mathit{path}\ x\ z\ \mathtt{R*add}(x,y,p,r) = \mathtt{Cons}(y, \mathit{path}\ x\ y\ p)$

Reasoning on this function requires a dependent elimination scheme

Dependent elimination scheme for inductive relations

$P(x,y,p)$ with $p : R^*(x,y)$

$$\frac{\begin{array}{l} f : \forall x.P(x,x,\mathtt{R*refl}(x)) \\ g : \ \forall x,y,z. \\ \qquad \forall p : R^*(x,y).P(x,y,p) \Rightarrow \forall r : R(y,z).P(x,z,\mathtt{R*add}(x,y,z,p,r)) \end{array}}{(\mathit{R*ind}\ f\ g) : \forall x,y.\forall p : R^*(x,y).P(x,y,p)}$$

Same reduction rules

Constructive interpretation of inductive relations

A relation may be defined inductively in many different ways

Example

$\texttt{sorted}(\texttt{Nil})$

$\forall a, l, m.\texttt{sorted}(l) \Rightarrow \texttt{sorted}(m)$

$\quad \Rightarrow \texttt{inflist}(a, m) \Rightarrow \texttt{suplist}(a, l)$

$\quad \Rightarrow \texttt{sorted}(\mathit{app}\ l\ \texttt{Cons}(a, m))$

Proving equivalence may be difficult
Each definition carry out a specific induction principle
A proof corresponds to a concrete representation
(prove transitivity of LE!)

Dependent types

Use inductive relations to represent more precise types.

- Lists of length n
- Terms of sort s

Example

$\texttt{Nil} : (\texttt{list}\ 0)$

$\texttt{Cons} : \forall n : \texttt{nat}.A \rightarrow (\texttt{list}\ n) \rightarrow (\texttt{list}\ n+1)$

No interest as a predicate on nat

$\mathit{map}_f : \forall n : \texttt{nat}.(\texttt{list}\ n) \rightarrow (\texttt{list}\ n)$

$\mathit{map}_f\ 0\ \texttt{Nil} = \texttt{Nil}$

$\mathit{map}_f\ (n+1)\ \texttt{Cons}(n, a, l) = \texttt{Cons}(n, f(a), \mathit{map}_f\ n\ l)$

$\mathit{app} : \forall n, m : \texttt{nat}.(\texttt{list}\ n) \rightarrow (\texttt{list}\ m) \rightarrow (\texttt{list}\ n+m)$

Dependent types : difficulties

Extra arguments of Cons, *map*, *app* for typing the real arguments ...
Propositional equality between terms with the same type
$(\mathit{app}\ \ n\ 0\ l\ \texttt{Nil}) = l$ not well-formed because
$(\mathit{app}\ \ n\ 0\ l\ \texttt{Nil}) : (\texttt{list}\ (n+0))$ and $l : (\texttt{list}\ n)$ not convertible
Requires more sophisticated equalities (dependent, John Major ...)
Cannot always replace a sub-term with a variable of the same type
Example $0 : \texttt{nat}$ in $(\mathit{app}\ \ n\ 0\ l\ \texttt{Nil})$.
Expect instance elimination scheme / partial pattern-matching

$$\frac{x : (\texttt{list}\ 0) \qquad f : P(\texttt{Nil})}{\textsf{match}\ x\ \textsf{with}\ \texttt{Nil} \rightarrow f : P(x)}$$

Mutual Inductive definition

A mutually inductive definition can be encoded as a single inductive family. An argument serves as an index.

$\texttt{I}_1(a_1 : A_1) \ldots \texttt{I}_n(a_n : A_n)$

$\texttt{c} : \forall x_1 : B_1 \ldots \forall x_l : B_l.\texttt{I}_i(t_i)$

Introduce a set of indexes X

$\texttt{inl}_i : A_i \rightarrow \texttt{X}$

$\texttt{I}(x : X)$

Replace in the constructors, $\texttt{I}_i(a_i)$ by $\texttt{I}(\texttt{inl}_i(a_i))$

Recursive definition of properties

Define a property $F\ n$ by induction on n

$$F\ 0 \Leftrightarrow G \quad F\ (n+1) \Leftrightarrow H(n, (F\ n))$$

H is stable with respect to equivalence :
$(P \Leftrightarrow Q) \Rightarrow (H(n, P) \Leftrightarrow H(n, Q))$
Inductive definition of "ϕ is a partial solution up to n"

$$\frac{\phi\ 0 \Leftrightarrow G}{(\texttt{partial}\ \phi\ 0)} \qquad \frac{(\texttt{partial}\ \phi\ n) \quad \phi\ (n+1) \Leftrightarrow H(n, (\phi\ n))}{(\texttt{partial}\ \phi\ n+1)}$$

Main property (induction on n):

$$(\texttt{partial}\ \phi\ n) \Rightarrow (\texttt{partial}\ \psi\ n) \Rightarrow (\phi\ n \Leftrightarrow \psi\ n)$$

Building the solution

Takes the union of all partial solutions (higher-order quantification)

$$\mathit{sol}\ n \equiv \exists \phi.(\texttt{partial}\ \phi\ n) \wedge \phi\ n$$

Check

$$\forall \phi.(\texttt{partial}\ \phi\ n) \Rightarrow ((\phi\ n) \Leftrightarrow (\mathit{sol}\ n))$$

Show $(\texttt{partial}\ \mathit{sol}\ n)$ by induction on n.

Inductive relations : summary

- A natural extension of the inductive type case
- Includes propositional equality
- Inversion schemes, partial pattern-matching are not primitive
- Even interpreted logically, carry a strong computational content

Inductive definitions in different systems

- Rules à la Martin-Löf
- Representation in second-order type theory
- Representation in higher-order logic
- Recursive definitions

Martin-Löf's Type Theory

Intensional Type Theory

- Constructive logic
- Propositions as type paradigm
- Logical connectives represented as inductive definitions
- Dependent elimination scheme
- Associated recursion operator
- Extension with universes, inductive-recursive schemes, records ...

Rules for primitive inductive definitions

$$t, T := \mathsf{prop} | x | \lambda x : T.t | \forall x : T.T |$$
$$\mathsf{Ind}(I : T)\{c_1 : T_1 | \dots | c_n : T_n\} | \mathsf{Constr}(i, T) | \mathsf{Elim}(T)(t_1, \dots, t_n)$$

Formation rule

$$\frac{(I : T \vdash T_1 : \mathsf{prop})_{i=1\dots n} \quad \text{positivity condition}}{\mathsf{Ind}(I : T)\{c_1 : T_1 | \dots | c_n : T_n\} : T}$$

Introduction rules

$$\frac{I \equiv \mathsf{Ind}(I : T)\{c_1 : T_1 | \dots | c_n : T_n\} \quad I : T \quad 1 \leq i \leq n}{\mathsf{Constr}(i, I) : T_i}$$

Elimination

Elimination rule

$$\frac{I \equiv \mathsf{Ind}(I : T)\{c_1 : T_1 | \dots | c_n : T_n\} \quad I : T \quad (f_i : \Phi(T_i))_{i=1\dots n}}{\mathsf{Elim}(I)(f_1, \dots, f_n) : \forall \vec{x}.\forall p : (I\ \vec{x}).P(\vec{x}, p)}$$

- Dependent/ Non dependent
- Weak/Strong
- One for both recursion and pattern-matching or one for pattern-matching and a fixpoint

Reduction rule

$$(\mathsf{Elim}(I)(f_1, \dots, f_n)\ \vec{t}\ (\mathsf{Constr}(i, I)\ \vec{a}))$$
$$\longrightarrow (f_i\ \phi(T_i, \mathsf{Elim}(I)(f_1, \dots, f_n), \vec{a}))$$

Martin-Löf's Type Theory

Related Implementations

- Nuprl, Cornell University
  - Extensional type theory
  - Partial terms
  - Subtyping ...
- ALF, Göteborg
  - Pattern-matching with dependent types + fixpoint
- Agda, Göteborg
  - Non-recursive inductive types (Case)
  - Fixpoint on proofs and types
  - No inductive relations (only fixpoints, ad-hoc equality)

Second-order encoding

System F (Girards, Reynolds)

Second order typed λ-calculus (minimal propositional logic)

Types $\quad T := \alpha \mid \forall\alpha.T \mid T \to T$

Terms $\quad t := x \mid \Lambda\alpha.t \mid \lambda x.t \mid (t\ T) \mid (t\ t)$

Typing $\quad \dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$

$$\frac{\Gamma \vdash t : T \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.T} \qquad \frac{\Gamma \vdash t : \forall\alpha.T}{\Gamma \vdash (t\ U) : T[\alpha := U]}$$

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda x : T.t : T \to U} \qquad \frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash t : T}{\Gamma \vdash (f\ t) : U}$$

Encoding inductive types in system F

Type : $\mathtt{I}$

Rules of construction : $c_1 : C_1, \ldots c_n : C_n$

$\mathtt{c}_i : \forall\alpha_1.\ldots\forall\alpha_l.T_1 \to \cdots \to T_m \to \mathtt{I}$

Definition : $\quad \mathtt{I} \equiv \forall\alpha.C_1[\mathtt{I} := \alpha] \to \cdots \to C_n[\mathtt{I} := \alpha] \to \alpha$

Examples

$\perp$		$\forall\alpha.\alpha$
$\top$	$() : \top$	$\forall\alpha.\alpha \to \alpha$
$\mathtt{bool}$	$\mathtt{true}, \mathtt{false} : \mathtt{bool}$	$\forall\alpha.\alpha \to \alpha \to \alpha$
$\mathtt{nat}$	$0 : \mathtt{nat}, \mathtt{S} : \mathtt{nat} \to \mathtt{nat}$	$\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$
$\beta * \gamma$	$\mathtt{P} : \beta \to \gamma \to \beta * \gamma$	$\forall\alpha.(\beta \to \gamma \to \alpha) \to \alpha$
$\beta + \gamma$	$\mathtt{inl} : \beta \to \beta + \gamma$	$\forall\alpha.(\beta \to \alpha) \to (\gamma \to \alpha) \to \alpha$
	$\mathtt{inr} : \gamma \to \beta + \gamma$	
$\exists\beta.T(\beta)$	$\mathtt{E} : \forall\beta.(T(\beta) \to \exists\beta.T)$	$\forall\alpha.(\forall\beta.(T(\beta) \to \alpha)) \to \alpha$

Deriving constructors and elimination

The non-recursive case

We have $\mathtt{I} \equiv \forall\alpha.C_1[\mathtt{I} := \alpha] \to \cdots \to C_n[\mathtt{I} := \alpha] \to \alpha$

We want $\mathtt{c}_i : C_i \equiv \forall\alpha_1.\ldots\forall\alpha_l.T_1 \to \cdots \to T_m \to \mathtt{I}$

$C_i[\mathtt{I} := \alpha] \equiv \forall\alpha_1.\ldots\forall\alpha_l.T_1 \to \cdots \to T_m \to \alpha$

Definition $c_i = \quad \Lambda\alpha_1.\ldots\Lambda\alpha_l.\lambda x_1 : T_1.\ldots\lambda x_m : T_m.$

$\qquad\qquad \Lambda\alpha.\lambda f_1 : C_1[\mathtt{I} := \alpha].\ldots\lambda f_n : C_n[\mathtt{I} := \alpha]$

$\qquad\qquad (f_i\ \alpha_1\ \cdots\ \alpha_l\ x_1\ \cdots\ x_m)$

Elimination

$\mathtt{I} \equiv \forall\alpha.C_1[\mathtt{I} := \alpha] \to \cdots \to C_n[\mathtt{I} := \alpha] \to \alpha$

Elimination just application, non-dependent

$$\frac{\Gamma \vdash p : \mathtt{I} \quad \Gamma \vdash f_1 : C_1[\mathtt{I} := C]\ \ldots\ \Gamma \vdash f_n : C_n[\mathtt{I} := C]}{\Gamma \vdash (p\ C\ f_1 \cdots f_n) : C}$$

Reduction

$c_i = \quad \Lambda\alpha_1.\ldots\Lambda\alpha_l.\lambda x_1 : T_1.\ldots\lambda x_m : T_m.$

$\qquad \Lambda\alpha.\lambda f_1, \ldots, f_n.(f_i\ \alpha_1\ \cdots\ \alpha_l\ x_1\ \cdots\ x_m)$

$(c_i\ A_1 \ldots A_l\ t_1\ \ldots t_m\ C\ f_1 \cdots f_n) \longrightarrow (f_i\ A_1 \ldots A_l\ t_1\ \ldots t_m)$

Examples

- Booleans $\mathtt{bool} = \forall\alpha.\alpha \to \alpha \to \alpha$
  - Constructors :

    $\mathtt{true} = \Lambda\alpha.\lambda x, y : \alpha.x \qquad \mathtt{false} = \Lambda\alpha.\lambda x, y : \alpha.y$
  - Conditional :

    $\mathit{if} : \forall\alpha.\alpha \to \alpha \to \mathtt{bool} = \Lambda\alpha.\lambda f, g : \alpha.\lambda b : \mathtt{bool}.(b\ \alpha\ f\ g)$
- $\beta * \gamma = \forall\alpha.(\beta \to \gamma \to \alpha) \to \alpha$
  - Constructor :

    $\mathtt{P} = \lambda x : \beta.\lambda y : \gamma.\Lambda C.\lambda f : \beta \to \gamma \to \alpha.(f\ x\ y) : \beta * \gamma.$
  - Projections :

    $\mathit{fst} : \beta * \gamma \to \beta = \lambda p.(p\ \beta\ \lambda x : \beta.\lambda y : \gamma.x)$

    $\mathit{snd} : \beta * \gamma \to \gamma = \lambda p.(p\ \beta\ \lambda x : \beta.\lambda y : \gamma.y)$

Recursive case

Condition Each recursive argument $T_i(\mathtt{I})$ should be **monotonic** :

$$\frac{f : X \to Y}{T_i[f] : T_i(X) \to T_i(Y)} \qquad \text{Allows } \mathtt{c} : ((\mathtt{I} \to T) \to U) \to \mathtt{I}$$

Typical case $\mathtt{c} : F(\mathtt{I}) \to \mathtt{I}$

Definition $\mathtt{I} = \forall\alpha.(F(\alpha) \to \alpha) \to \alpha$

Constructor

$$\mathtt{c} = \lambda a : F(\mathtt{I}).\Lambda\alpha.\lambda f : F(\alpha) \to \alpha.(f\ (F[\lambda x : \mathtt{I}.(x\ \alpha\ f)]\ a))$$

Typing

$$\dfrac{\dfrac{\lambda x : \mathtt{I}.(x\ \alpha\ f) : \mathtt{I} \to \alpha}{F[\lambda x : \mathtt{I}.(x\ \alpha\ f)] : F(\mathtt{I}) \to F(\alpha)} \qquad a : F(\mathtt{I})}{(F[\lambda x : \mathtt{I}.(x\ \alpha\ f)]\ a) : F(\alpha)}$$

Recursive case, elimination

Iterative non-dependent scheme

$$\frac{p : \mathtt{I} \qquad f : f(T) \to T}{(p\ T\ f) : T}$$

Example : natural numbers

- $\mathtt{nat} = \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$
- Constructors : $\mathtt{0} = \Lambda\alpha.\lambda x : \alpha.\lambda f : \alpha \to \alpha.x$

  $\mathtt{S} = \lambda n : \mathtt{nat}.\Lambda\alpha.\lambda x : \alpha.\lambda f : \alpha \to \alpha.x$
- Iterator : $\mathit{it} = \Lambda\alpha.\lambda x : \alpha.\lambda f : \alpha \to \alpha.\lambda n : \mathtt{nat}.(n\ \alpha\ x\ f)$

$$\begin{cases} (\mathit{it}\ T\ g\ h\ \mathtt{0}) & = g \\ (\mathit{it}\ T\ g\ h\ (\mathtt{S}\ n)) & = (h\ (\mathit{it}\ T\ g\ h\ n)) \end{cases}$$

Impredicative encodings : problems

- $\mathtt{true} \neq \mathtt{false}$ not provable (proof-irrelevance model)

  but $\mathtt{true} \neq \mathtt{false}$ implies non-confusion $c_i(\vec{x}) \neq c_j(\vec{y})$
- How to write the predecessor function ?

  Fixpoint theorem $(\mathtt{nat} \to 1 + \mathtt{nat})$

  or standard trick to derive recursion from iteration :

$$\begin{cases} (\mathit{f\_it}\ \mathtt{0}) & = g \\ (\mathit{f\_it}\ (\mathtt{S}\ n)) & = (h\ (\mathit{f\_it}\ n)) \end{cases}$$

$$\begin{cases} (\mathit{f\_rec}\ \mathtt{0}) & = g \\ (\mathit{f\_rec}\ (\mathtt{S}\ n)) & = (h\ n\ (\mathit{f\_rec}\ n)) \end{cases}$$

$$\begin{cases} (\mathit{f'\_rec}\ \mathtt{0}) & = (\mathtt{0}, g) \\ (\mathit{f'\_rec}\ (\mathtt{S}\ n)) & = \mathsf{let}\ (n, x) = (\mathit{f'\_rec}\ n)\ \mathsf{in}\ ((\mathtt{S}\ n), (h\ n\ x)) \end{cases}$$

Predecessor function

Specification $\quad pred\ 0 = 0 \quad pred\ (n+1) = n$

Definition
$$\begin{cases} pred'\ 0 & = (0,0) \\ pred'\ n+1 & = \mathsf{let}\ (m,x) = pred'\ n\ \ \mathsf{in}\ (m+1,m) \\ pred\ n & = \mathsf{let}\ (m,x) = pred'\ n\ \ \mathsf{in}\ x \end{cases}$$

Reduction

$$\begin{array}{lllll} pred\ 3 & \longrightarrow \mathsf{let}\ (m,x) = pred'\ 3 & \mathsf{in}\ x & \longrightarrow 2 \\ pred'\ 3 & \longrightarrow \mathsf{let}\ (m,x) = pred'\ 2 & \mathsf{in}\ (m+1,m) & \longrightarrow (3,2) \\ pred'\ 2 & \longrightarrow \mathsf{let}\ (m,x) = pred'\ 1 & \mathsf{in}\ (m+1,m) & \longrightarrow (2,1) \\ pred'\ 1 & \longrightarrow \mathsf{let}\ (m,x) = pred'\ 0 & \mathsf{in}\ (m+1,m) & \longrightarrow (1,0) \\ pred'\ 0 & \longrightarrow (0,0) \end{array}$$

Representation problems

- $pred\ n$ requires n steps of computation
- $pred\ (n+1)$ and n are in general non equivalent on open terms
- there are closed normal terms in $\beta * \gamma$ which are not of the form $(\mathtt{P}\ b\ c) = \Lambda\alpha.\lambda f : \beta \to \gamma \to \alpha.(f\ b\ c)$
  Assume $\beta = \gamma = \bot \to \bot$
  (negative universal quantification, $\top \to \top$ works as well)
  We have $id = (\lambda x : \bot.x) : \beta$
  $\Lambda\alpha.\lambda f : \beta \to \gamma \to \alpha.(f\ (\lambda x : \bot.(x\ \alpha \to \bot\ (f\ id\ id)))\ id)$

Extensions with fixpoints

- Add fixpoint combinator to improve intensional behavior
  - Parigot, Krivine, Audebaud ...
    $\mu\alpha.F(\alpha) = F(\mu\alpha.F(\alpha))$
    $nat = \forall\alpha.\alpha \to (nat \to \alpha \to \alpha) \to \alpha$
  - Mendler : $\mu = \mu\alpha.F(\alpha) \qquad \mathtt{c} : F(\mu) \to \mu$
    $$\frac{\phi : \forall\beta.(\beta \to \mu) \to (\beta \to \gamma) \to F(\beta) \to \gamma}{\mathsf{rec}\ \phi : \mu \to \gamma} \qquad F(\mu * \gamma) \to \gamma$$
    $\mathsf{rec}\ \phi\ (\mathtt{c}\ p) \longrightarrow (f\ \mu\ id\ (\mathsf{rec}\ \phi)\ p)$
    Example $nat = \mu\alpha.(1 + nat) = \mu\alpha.(\forall\beta.\beta \to (\alpha \to \beta) \to \beta)$
    $$\begin{cases} f\ 0 & = g \\ f(n+1) & = h(n, f(n)) \\ (\phi\ \beta\ p\ r\ n) & = (n\ \gamma\ g\ \lambda x : \beta.h((p\ x),(r\ x))) \end{cases}$$

Extensions with dependent types

Extends system F with higher-order type schemes (F_ω) or dependent types (Calculus of Constructions).

The same scheme applies.

- $\mathtt{I} : \kappa$ with κ an arity $(\forall z_1 : s_1 \ldots. \forall z_k : s_k.\mathrm{Prop})$
  Examples : $\mathrm{Prop} \to \mathrm{Prop}$ or $\mathtt{nat} \to \mathrm{Prop}$ ...
- rules of constructions $c_1 : C_1, \ldots, c_n : C_n$
  $C_i \equiv \forall x_1 : T_1 \ldots. \forall x_m : T_m.(\mathtt{I}\ t_1 \ldots t_k)$
  Definition $\mathtt{I}$:
  $\lambda z_1 : s_1 \ldots \lambda z_k : s_k.$
  $\forall\alpha : \kappa.C_1[\mathtt{I} := \alpha] \to \cdots \to C_n[\mathtt{I} := \alpha] \to (\alpha\ z_1 \cdots z_k)$

Good representation for inductive relations but not for types.

Higher-order logic

HOL, Isabelle/HOL, PVS ...

- Objects : simply typed λ-calculus with a type of propositions o, an (infinite) type of individuals ι. Any type is inhabited.
- Higher-order logic $\forall x : \tau . p(x)$
  covers quantification on objects $\forall x : \iota . p(x)$, $\forall x : \iota \to \iota . p(x)$
  and higher-order quantification on propositions
  $\forall x : o . p(x)$, $\forall x : \iota \to \iota \to o . p(x)$
- Classical logic $o =$ `bool`, functions as relations.

Representation in higher-order logic

- Description operator $\epsilon x . P(x)$.
  From a relational function $F : \alpha \to \beta \to o$
  to a term $\lambda x . \epsilon y . F(x, y) : \alpha \to \beta$
- Type restriction : from predicates to new types.

$$\tau \underset{\text{abs}}{\overset{\text{rep}}{\rightleftarrows}} \{x : \sigma | P(x)\}$$

Properties of HOL

- The object language cannot be extended with polymorphic quantification.
- The object language can be extended with (primitive) (strictly positive) inductive definitions (ECC, Luo, Goguen).
- Logical connectives, inductive relations can be defined with higher-order quantification.

Product definition : $\alpha * \beta$

- Take the type of binary relations $X = \alpha \to \beta \to o$
- Define the pair $(a, b) = \lambda x : \alpha . \lambda y : \beta . x = a \wedge y = b$
- Define the set of relations which represents pairs $\mathit{Pair}(P)$:
  $\exists a : \alpha . \exists b : \beta . P(a, b)$
  $\wedge\ \forall a_1, a_2 : \alpha . \forall b_1, b_2 : \beta . P(a_1, b_1) \Rightarrow P(a_2, b_2) \Rightarrow a_1 = a_2 \wedge b_1 = b_2$
- Takes $\alpha * \beta$ be the type of predicates P which satisfies $\mathit{Pair}(P)$.
- Check (a, b) satisfies Pair.
- First projection $\mathit{fst} : \alpha * \beta \to \alpha$ defined as a relation :
  $\mathit{Fst}(p, x) = \exists y : \beta . \mathrm{rep}(p)(x, y)$.

Inductive types in HOL

T. Melham, E. Gunter, L. Paulson, J. Harrison...

The key steps :

- Build a type X, such that one can build injective functions for the constructors.
  $z : X \quad s : X \to X \quad s(x) = s(y) \to x = y \quad s(x) \neq 0$
- Define by induction the smallest subset $I(x)$ of X closed by the rules of construction.
  $N(x) = \forall P : X \to o.P(z) \to (\forall y.P(y) \to P(s(y))) \to P(x)$
- Define $\mathtt{I}$ as the restriction of X to objects which satisfy I.
  $\mathtt{abs} : X \to \mathtt{I} \quad \mathtt{rep} : \mathtt{I} \to X$
  $\mathtt{abs}(\mathtt{rep}(n)) = n \quad I(x) \Rightarrow \mathtt{rep}(\mathtt{abs}(x)) = x \quad I(\mathtt{rep}(n))$

Properties of the inductive type

- Define constructors with the apropriate type using abs and rep.
  $\mathtt{0} = \mathtt{abs}(z) \quad \mathtt{S}(n) = \mathtt{abs}(s(\mathtt{rep}(n)))$
- $\mathtt{S}(n) = \mathtt{S}(m) \quad s(\mathtt{rep}(n)) = s(\mathtt{rep}(m)) \quad \mathtt{rep}(n) = \mathtt{rep}(m) \quad n = m$
- $\mathtt{S}(n) = \mathtt{0} \quad s(\mathtt{rep}(n)) = z \quad \perp$
- Derive induction principle for $\mathtt{I}$ using property I.

$$\frac{P(\mathtt{0}) \qquad \forall n : \mathtt{N}.P(n) \Rightarrow P(\mathtt{S}(n))}{\forall n : \mathtt{N}.P(n)}$$

Show $\forall x.N(x) \to P(\mathtt{abs}(x))$ and use

Recursion scheme

Prove the existence of a general recursor
$\forall g : \alpha.\forall h : \mathtt{N} \to \alpha \to \alpha.$
$\exists! f : \mathtt{N} \to \alpha.f(\mathtt{0}) = g \wedge \forall n.f(\mathtt{S}(n)) = h(n, f(n))$

Define $F : \mathtt{N} \to \alpha \to o$ inductively :

$$\frac{}{F(\mathtt{0}, g)} \qquad \frac{F(n, a)}{F(\mathtt{S}(n), h(n, a))}$$

Prove $\forall n : \mathtt{N}.\exists a : \alpha.F(n, a)$ and
$\forall n : \mathtt{N}.\forall a, b : \alpha.F(n, a) \Rightarrow F(n, b) \Rightarrow a = b$ and

Take $f(n)$ be $\epsilon a.F(n, a)$.

Computation done by equational reasoning.

Building the apropriate support

- A large enough tree set, built from ι,o and function space.
- Existence of injection, using cardinality.
- Monotonic (non strictly) positive inductive types are not legal.

$$(\mathtt{B} \to o) \xrightarrow{\lambda p,q.p=q} ((\mathtt{B} \to o) \to o) \underset{\mathtt{out}}{\overset{\mathtt{in}}{\rightleftarrows}} \mathtt{B}$$

Injective function $f : (\mathtt{B} \to o) \to \mathtt{B}$
$P_0(x) = \exists P.f(P) = x \wedge \neg P(x) \quad i_0 = f(P_0)$

$$P_0(i_0) \Leftrightarrow \neg P_0(i_0)$$

Recursive definitions

Classical technics

- Primitive recursive scheme
- Well-founded recursion
- Compile other schemes to one of these

p

Recursive definitions

More specific solutions

- Syntactic criteria for termination (Coq)
- Capture structural fixpoint by typing
  Mendler, Giménez, Barthe, Amadio, Altenkirch ...
  $$\frac{f : \texttt{nat}^{x} \to \alpha \vdash t : \texttt{nat}^{s(x)} \to \alpha}{\textsf{let rec } f\ x = t : \texttt{nat} \to \alpha}$$
- Introduce rewriting in conversion
  - General scheme (Blanqui, Jouannaud, Okada)
  - RPO (Walukiewicz, Jouannaud, Rubio)

  Normalisation, confluence, consistency, completeness, efficiency of reduction

Other extensions

- Record
- Induction-recursion
- Quotient types
- Universes

Applications

- Deep embedding of programming languages
- Shallow embedding of programs
- Proof of cryptographic protocols
- Proof by reflexion

Deep embedding

Objectives :

- Study the properties of a programming language
- Tools working on this programming language

Examples :

- Properties of the Java(Card) Virtual Machine
  - Byte-code verifier
  - Offensive versus defensive machine
  - Firewall

Deep-embedding : principles

Introduce an inductive type to represent the syntax of programs.

- Abstract syntax tree
- Representation of bounded variables/substitution
  Different solutions
  - De Bruijn indexes
  - Explicit substitution
  - Higher-order abstract syntax (`lambda` → `lambda`)
- Functions on terms
- Semantics as inductive relations or functions

Current experiments (Java(Card) programs)

- BALI project in Munich (T. Nipkow, `http://isabelle.in.tum.de/Bali/`) - Isabelle/HOL
  Type safety of the Java source language and part of the JVM
  Axiomatic semantic of Java
- FORMAVIE (Trusted Logic, Schlumberger) - Coq
  Modelisation of security properties of the whole Card system
  Several applets may run on the card preserving confidentiality
- Jakarta (INRIA Sophia-Antipolis) - Coq Public semantics of JavaCard Virtual machine, byte-code verifier
  Functional representation of execution steps
  On well-typed programs, defensive and offensive machine coincide

Expected properties of inductive definitions

- Efficient behavior with a large number of rules of construction
- Executable specifications
  - TWELF, `http://www-2.cs.cmu.edu/~twelf/`
  - Functional representation (Jakarta)

Shallow embedding

Interpret a program in the proof assistant by its semantics

- The LOOP project (B. Jacobs, Nijmegen,PVS)
- Jive (P. Poetzsch-Heffter, Kaiserslautern, Isabelle/HOL)
- Jack (J.-L. Lanet,Gemplus , B)
- Krakatoa (C. Marché, Orsay,Coq)

Use concrete data-structures, functions

A cryptographic protocol

(L. Paulson in Isabelle/HOL)

A system of paid TV

- The decoder n asks the smart-card n whether the fees have been paid.
  1. $D_n \longrightarrow C_n : (D_n, m_1)$
  2. $C_n \longrightarrow D_n : (C_n, m_2)$
- They share a secret (encrypted) information m_1, m_2.
- Every exchanged message is public.

Modelisation

Inductive definitions of agents and messages

```
Inductive agent : Set
   := C : nat→agent | D : nat→agent | I : agent.

Variables key,data : Set.

Inductive message : Set :=
        Name  : agent → message
      | K    : key → message
      | Dat  : data → message
      | Enc : message → key → message
      | P   : message → message → message.
```

Protocol

Parameterised by the messages m_1, m_2 and a predicate *paid* such that (*paid* n) true when the consumer n has paid the fees.

Three predicates mutually defined :

```
send : agent → message → Prop
receive : agent → message → Prop
known : message → Prop
```

`send` follows the protocol, but the intruder may send any message he knows.
Anybody receives any sent message
The intruder knows any message he received, or any public knowledge or what can be deduced from that.

Definitions send and receive

```
Inductive send  : agent → message → Prop :=
     init   : ∀ n:nat.(send (D n) (P (Name (D n)) m1))
   | trans1 : ∀ n:nat.(receive (C n) (P (Name (D n)) m1)) → (pa
                  → (send (C n) (P (Name (C n)) m2))
   | cheat : ∀ m:message.(known m) → (send I m)

with receive : agent → message → Prop :=
     link : ∀ m:message.(A,B:agent) (send A m)→(receive B m)

Definition ok [n:nat] : Prop
   := (receive (D n) (P (Name (C n)) m2)).
```


Definition known

```
known : message → Prop :=
    spy: ∀ m:message.(receive I m)→(known m)
  | name: ∀ a:agent.(known (Name a))
  | decomp_l: ∀ m1,m2:message.(known (P m1 m2))→(known m1)
  | decomp_r: ∀ m1,m2:message.(known (P m1 m2))→(known m2)
  | compose: ∀ m1,m2:message.
              (known m1)→(known m2)→(known (Pair m1 m2))
  | crypt : ∀ m:message. ∀ k:key.
             (known m)→(known (K k))→(known (Enc m k))
  | decrypt : ∀ m:message. ∀ k:key.
               (known (Enc m k))→(known (K k))→(known m).
```


Expected properties

The protocol works properly if the fee has been paid

```
Lemma paid_get :
   ∀ m1,m2:message.∀ n:nat.(paid n) → (ok m1 m2 n).
```

First flaw when only one message

```
Lemma flaw : ∀ m:message. ∀ n:nat.(ok m m n).
```

1. $D_n \longrightarrow I : (D_n, m)$
2. $I \longrightarrow D_n : (C_n, m)$

Proving correctness

No flaw when nobody has paid.

```
Definition m1 := (Enc (Data d1) k).
Definition m2 := (Enc (Data d2) k).
Hypothesis no_paid : ∀ n:nat.¬(paid n).
Hypothesis d1_not_d2 : ¬(d1=d2).
Lemma no_flaw : ∀ n:nat.¬(ok m1 m2 n).
```

Invariant

```
Inductive invknown :  message → Prop :=
   base : ∀ a:agent.(invknown (Name a))
 | mes  : (invknown m1)
 | invcomp : ∀ m1,m2:message.
       (invknown m1)→ (invknown m2)→(invknown (P m1 m2)).
```


Properties

```
Lemma invknown_not_ok
   : ∀ n:nat.¬(invknown (P (Name (C n)) m2)).
```

by inversion of `invknown` using the shape of $m2$.

```
Lemma no_flaw
   : ∀ n:nat.(ok m1 m2 n)→(invknown (P (Name (C n)) m2)).
```

by mutual induction, anything which is sent or received satisfies `invknown`.

A second flaw

```
Variable p:nat.
Hypothesis p_paid : (paid p).

Lemma flaw2 : ∀ n:nat. (ok m1 m2 n).
```

Trace of the proof

$$\begin{array}{lll} D_p & \longrightarrow C_p & :(D_p, m_1) \\ C_p & \longrightarrow D_p, I & :(C_p, m_2) \\ D_n & \longrightarrow C_n & :(D_n, m_1) \\ I & \longrightarrow D_n & :(C_n, m_2) \end{array}$$

Inductive definitions in this example

- Limit the knowledge of the intruder
- Correctness with respect to a certain model
- Automatic proof-tool for finding flaws
- Use proof-term as trace of execution

Proofs by reflexion

Internalizing a proof procedure.

Example

Simplify expressions with addition and numerals on natural numbers :

$$e \longrightarrow e_1 + \cdots + e_n + n$$

with e_i an expression which is not an addition nor a numeral and n a numeral.

Require tedious steps of rewriting on $+$.

Reflexion strategy

- lift e to a concrete expression e'
- program a normalisation function *norm* on concrete expressions
- compute (*norm* e') to an expression e''
- show (*norm* e') and e'' represents the same natural number.

Consequences

- Meta-strategy programming becomes *norm* definition
- General proof of correctness of *norm*
- Reasoning steps are replaced by computation on *norm*
- Introduce intermediate representation

Reflexion : quoting

```
Inductive aexpr : Set :=
    Var : nat → aexpr
  | Plus : aexpr → aexpr → aexpr
  | Num : nat → aexpr.

Fixpoint interp [v:nat→ nat; e:aexpr] : nat :=
   Cases e of (Var x) => (v x)
            | (Plus e1 e2) => (interp v e1)+(interp v e2)
            | (Num n) => n
   end.
```


Reflexion : normalisation

```
Fixpoint flat [e:aexpr] : nat * (list nat) :=
 Cases e of (Var x) => (O,(cons x (nil nat)))
    | (Plus e1 e2) => let (n1,l1) = (flat e1) in
      let (n2,l2) = (flat e2) in ((n1+n2),(app l1 l2))
    | (Num n) => (n,(nil nat))
 end.
Fixpoint exprlv [n:nat; l: (list nat)] : aexpr :=
 Cases l of nil => (Num n)
   | (cons x m) => (Plus (Var x) (exprlv n m))
 end.
Definition norm [e:aexpr] : aexpr :=
    let (n,lv)=(flat e) in (exprlv n lv).
```

Reflexion : Correctness and usage

```
Lemma flat_correct : ∀ v:nat→ nat.∀ e:aexpr.
    (interp v e)=(interp v (norm e)).
```

Usage

```
Variables x,y,z:nat.
Lemma test : ((5+x)+1)+((y+2)+((x*y)+z))=0

Definition v:= [n]Cases n of
   0 ⇒ x | 1 ⇒ y | 2 ⇒ z | 3 ⇒ (x * y) | _ ⇒ 0
end.
```


Reflexion : Usage

```
Lemma test :((5+x)+1)+((y+2)+((x*y)+z))=0

Change (interp v
        (Plus (Plus (Plus (Num 5) (Var 0)) (Num 1))
        (Plus (Plus (Var 1) (Num 2))
              (Plus (Var 3) (Var 2)))))=0.
Rewrite flat_correct; Simpl.
```

Result

$$x + (y + (x * y + (z + 8))) = 0$$

Expected properties of inductive definitions

- Good data-structures
- Efficient reduction strategy
- Automate reasoning on programs

Inductive definitions : summary

- A high-level scheme useful in many contexts
- Non recursive data-types related to logic
- Strong relationship between induction and recursion
- Strong rules may lead to inconsistency
- Encodings in HOL and type theory