Applications of the Lambda Calculus

Representing computable functions and proofs

Henk Barendregt

Nijmegen University

The Netherlands

1 Computability

Abstract syntax for lambda terms

atom = a | atom ´

term = atom | term term | λ atom term

Axiom

$(\lambda x.M)\ N = M\ [x{:=}N]$ (β)

taken for granted identifications like

$\lambda x.x = \lambda y.y$ (α)

atoms $a, a', a'', \ldots$

we write $x, y, z, \ldots$

terms $x, \lambda x.x, (\lambda x.x)y, \ldots$

we write $M, N, L, \ldots$

Intended interpretation

$\lambda x.M(x)$ function F such that $F: x \mapsto M(x)$

$F\,x$ F applied to x

hence for $F = \lambda x.M(x)$

$$F\,x = M(x)$$

$$F\,N = M(N) \quad = M\,[x{:=}N]$$

Notational convention

$$\boxed{\begin{aligned} &M\,N_1 \ldots N_k = (..((MN_1)N_2) \ldots N_k) \\ &\lambda x_1 \ldots x_k.M = (\lambda x_1(\lambda x_2 \ldots (\lambda x_k\,M)..)) \end{aligned}}$$

Then using k times (β) one can deduce

$$\boxed{(\lambda x_1 \ldots x_k.M(x_1,\ldots,x_k))\,X_1 \ldots X_k = M(X_1,\ldots,X_k)}$$

Representing numbers (Church's numerals)

$$c_n = \lambda fx.f^n x$$

where $f^0 x = x$ and $f^{n+1}x = f(f^n x)$

Representing computable functions

Simple functions (Rosser 1935)

$$S^+ \, c_n = c_{n+1}$$
$$A_+ \, c_n \, c_m = c_{n+m}$$
$$A_* \, c_n \, c_m = c_{n.m}$$
$$A_{exp} \, c_n \, c_m = c_{n\hat{}m}$$

for

$$S^+ = \lambda n.(\lambda fx.f(nfx))$$
$$A_+ = \lambda nm.(\lambda fx.nf(mfx))$$
$$A_* = \lambda nm.(\lambda fx.n(mf)x)$$
$$A_{exp} = \lambda nm.(\lambda fx.mnfx)$$

We say that the function $f:\mathrm{IN}^k \to \mathrm{IN}$ is represented by the lambda term F iff

$$F \, c_{n_1} \ldots c_{n_k} = c_{f(n_1,\ldots,n_k)}$$

Standard terms

$\text{true} = \lambda pq.p$	(boolean)
$\text{false} = \lambda pq.q$	(boolean)
$\text{if } B \text{ then } P \text{ else } Q = B\ P\ Q$	(conditional)
$[M,N] = \lambda z.z\ M\ N$	(pairing)
$\langle M_1,\ldots,M_n\rangle = \lambda z.z\ M_1 \ldots M_n$	(n-tuple)
$\text{zero}_? = \lambda n.(n\ (\lambda x.\text{false}))\ \text{true}$	(test for zero)

Note that

$$\text{if true then } P \text{ else } Q = P$$

$$\text{if false then } P \text{ else } Q = Q$$

$$[M,N]\ \text{true} = M \text{ and } [M,N]\ \text{false} = N$$

$$\text{zero}_?\ c_0 = \text{true}$$

$$\text{zero}_?\ c_{k+1} = \text{false}$$

Write $X.1 = X\ \text{true}$ and $X.2 = X\ \text{false}$ then

$$[M_1,M_2].i = M_i \qquad \text{for } i=1,2.$$

More generally write $P_{n,i} = \lambda x_1 \ldots x_n.x_i$. Then

$$\langle M_1,\ldots,M_n\rangle\ P_{n,i} = M_i,$$

Fixed-point theorem

For all F there exists an X such that $FX=X$

Proof Take $W=\lambda x.F(x\ x)$ and $X=W\ W$. Then

$$X = W\ W = (\lambda x.F\ (x\ x))\ W = F(W\ W) = FX \ \blacksquare$$

Corollary Given a term $G=G(h,x_1,\ldots,x_n)$. Then there is a term H such that

$$H\ x_1 \ldots\ x_n = G(H,x_1,\ldots,x_n)$$

Proof Apply the fixedpoint theorem to

$$F = \lambda h x_1 \ldots\ x_n\ G(h,x_1,\ldots,x_n) \ \blacksquare$$

Theorem (Kleene 1936) All computable functions on IN can be represented this way

Proof The initial functions

$$\mathrm{zero}(x) = x+1$$

$$\mathrm{succ}(x) = x+1$$

$$U_{n,i}\ (x_1,\ldots,x_n) = x_i$$

can be represented respectively by

$$\lambda n.c_0,\ S^+ \text{ and } P_{n,i} = \lambda x_1 \ldots x_n.x_i$$

Functions obtained by primitive recursion can be represented as follows. Let

$$f(0) = 13$$

$$f(k+1) = h(f(k),k)$$

and suppose that h is represented by H.

We want to represent pairs $(k,f(k))$

Note that

$$T\ [c_k, c_{f(k)}] = [c_{k+1}, c_{f(k+1)}]$$

where

$$T = \lambda p\ .[S^+\ p.1,\ H\ p.2\ p.1]$$

Then

$$T^k\ [c_0,\ c_{13}] = [c_k,\ c_{f(k)}]$$

so $F = \lambda k\ .(k\ T\ [c_0,\ c_{13}]).2$ does the job.

Functions defined by minimalization can be represented as follows. Let

$$g(n) = \mu x[h(x,n)=0]$$

and suppose h is represented by H. Let

$$\begin{array}{llll} F\ n\ x & = & x & \text{if } H\ n\ x = c_0 \\ & = & F\ n\ (S^+\ x) & \text{else} \end{array}$$

[Use corollary:

$F\ n\ x = \text{if } (\text{zero}_?\ (H\ n\ x)) \text{ then } x \text{ else } (F\ n\ (S^+\ x))$]

Then we can take as representation for g

$$G = \lambda n.F\ n\ c_0$$

Indeed

$$\begin{array}{lll} G\ n & = & F\ n\ c_0 \\ & = & F\ n\ c_1 \\ & \ldots\ldots & \\ & = & F\ n\ c_k \\ & = & c_k \end{array}$$

as soon as $H\ n\ c_k$ equals zero. ■

Algebraic data types

nat = zero | succ nat

tree = bud | leaf nat | tree + tree

(binary labelled trees)

+   +

3   +   +   3

5   •   •   5

Representation of computable functions over data types

(Böhm-Piperno-Guerini 1993)

Define ψ: tree $\rightarrow$ term as follows

$$\boxed{\begin{aligned} \psi(\bullet) &= \lambda e.e\, P_{3,1}\, e \\ \psi(\text{leaf } n) &= \lambda e.e\, P_{3,2}\, n\, e \\ \psi(t_1+t_2) &= \lambda e.e\, P_{3,3}\, \psi(t_1)\, \psi(t_2)\, e \end{aligned}}$$

This can best be remembered by taking as representation of the constructors

B	=	$\lambda e.e\, P_{3,1}\, e$
L	=	$\lambda n\, \lambda e.e\, P_{3,2}\, n\, e$
P	=	$\lambda t_1 t_2\, \lambda e.e\, P_{3,3}\, t_1\, t_2\, e$

<u>Theorem</u> Given terms A_0, A_1, A_2 there exists a term F such that

F B	=	$A_0\, F$
F (L n)	=	$A_1\, n\, F$
F(P t_1 t_2)	=	$A_2\, t_1\, t_2\, F$

<u>Proof</u> Try $F = \langle\langle X_0, X_1, X_2\rangle\rangle$. Then we want

$$
\begin{aligned}
F\,B &= B\,\langle X_0, X_1, X_2\rangle \\
&= P_{3,1}\, X_0\, X_1\, X_2\, \langle X_0, X_1, X_2\rangle \\
&= X_0\, \langle X_0, X_1, X_2\rangle \\
&= A_0\, \langle\langle X_0, X_1, X_2\rangle\rangle \\
&= A_0\, F,
\end{aligned}
$$

which holds if we take $X_0 = \lambda x.A_0 \langle x\rangle$.

Similarly we can find X_1 and X_2. ■

Self-interpretation

Consider the data type with constructors

const	(unary)
app	(binary)
abs	(unary)

<u>Proposition</u> (Mogensen 1992)

Define #: term $\rightarrow$ term

#(x)	=	const x
#(P Q)	=	app #(P) #(Q)
#(λx.P)	=	abs (λx.#(P))

Then there exists a self-interpreter E such that for all terms M

E #(M) = M

<u>Proof</u> By the construction of Böhm et al there is an E satisfying

E (const x)	=	x
E (app p q)	=	(E p) (E q)
E (abs z)	=	λx.E (z x)

The statement follows by induction on M.■

We can take E = <<K,S,C>>.

Exercises

1.1 The reduction graph of a term is

$$G_\beta(M) = (\{N \mid M \twoheadrightarrow_\beta N\}, \rightarrow_\beta).$$

Draw $G_\beta(WWW)$ with $W = \lambda xy.xyy$.

1.2 Prove that A_{exp} represents exponentiation.

1.3 Represent $f(x)=x!$, the factorial.

1.4 Represent on trees g_{mir} (mirroring) such that e.g.

$$g_{mir}(\text{leaf}(3) + ((\text{leaf}(5) + \bullet)) = (\bullet + \text{leaf}(5)) + \text{leaf}(3)$$

1.5 Represent a function on trees that squares the numbers at the nodes and adds them.

1.6 (i) Define a term fst such that

$$\begin{array}{lll} \text{fst}\ \#(M) & = & P \quad \text{if } M \text{ is } (P\ Q) \\ & = & \text{false} \quad \text{else} \end{array}$$

(ii) Show that there is no term F such that

$$F\,M = P \quad \text{if } M \text{ is } (P\ Q).$$

2 Functional programming

Types are like dimensions in physics

They provide partial correctness

typevar = p | typevar ´
type = typevar | type $\to$ type

(Typing) statement

$M : \sigma$ "M is of type τ, M in τ"

Context

$\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$

Intuitive type assignment

$f : \sigma \to \tau, x : \sigma \vdash f\, x : \tau$

Type vars: $p, p', p'', \ldots$

in general: $q, r, \ldots$

Types: $p \to p$, $p \to (q \to p)$, $p \to q$, ...

in general: $\sigma, \tau, \ldots$

Notations

$\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$ stands for

$(\sigma_1 \to (\sigma_2 \to (\ldots \to \sigma_n)..))$

$\vdash M : \sigma$ stands for $\varnothing \vdash M : \sigma$

$\Gamma, x{:}\tau \vdash M : \sigma$ stands for $\Gamma \cup \{x{:}\tau\} \vdash M : \sigma$

Formal system $\lambda\!\to$ of type assignment

$\lambda\!\to$ implicit (Curry) version

$$(x : \sigma) \in \Gamma \Rightarrow \Gamma \vdash x : \sigma$$

$$\Gamma \vdash F : (\sigma \to \tau),\ \Gamma \vdash a : \sigma \Rightarrow \Gamma \vdash (F\, a) : \tau$$

$$\Gamma, x : \sigma \vdash M : \tau \Rightarrow \Gamma \vdash (\lambda x.M) : (\sigma \to \tau)$$

$\lambda\!\to$ explicit (Church) version

$$(x : \sigma) \in \Gamma \Rightarrow \Gamma \vdash x : \sigma$$

$$\Gamma \vdash F : (\sigma \to \tau),\ \Gamma \vdash a : \sigma \Rightarrow \Gamma \vdash (F\, a) : \tau$$

$$\Gamma, x : \sigma \vdash M : \tau \Rightarrow \Gamma \vdash (\lambda x{:}\sigma.M) : (\sigma \to \tau)$$

Examples Define

$I_\sigma = \lambda x{:}\sigma.x$

$K_{\sigma\tau} = \lambda x{:}\sigma\lambda y{:}\tau.x$

$S_{\sigma\tau\rho} = \lambda x{:}\sigma\to\tau\to\rho\lambda y{:}\sigma\to\tau\lambda z{:}\sigma.xz(yz)$

Then

$$\vdash I_\sigma : \sigma\to\sigma$$

$$\vdash K_{\sigma\tau} : \sigma\to\tau\to\sigma$$

$$\vdash S_{\sigma\tau\rho} : (\sigma\to\tau\to\rho)\to(\sigma\to\tau)\to\sigma\to\rho$$

Non-empty context

$$x{:}\sigma \vdash I_\sigma\, x : \sigma$$

Substitution theorem

$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma^* \vdash M : \sigma^*$ $\quad *$ is a substitution

$\Gamma, x{:}\sigma \vdash M : \tau \;\&\; \Gamma \vdash N : \sigma \;\Rightarrow\; \Gamma \vdash M[x{:=}N] : \tau$

Subject reduction theorem

$\Gamma \vdash M : \sigma \;\&\; M \twoheadrightarrow_\beta M' \;\Rightarrow\; \Gamma \vdash M' : \sigma$

Strong normalization theorem

$\Gamma \vdash M : \sigma \;\Rightarrow\;$ M is strongly normalizing

Principal type theorem (Curry version)

[Curry, Hindley 1969]

If M is typable, then M has a principle pair (Γ_0,σ_0):

$\Gamma_0 \vdash M : \sigma_0$

$\Gamma \vdash M : \sigma \;\Rightarrow (\Gamma,\sigma) = (\Gamma_0,\sigma_0)*$ $\quad *$ is a substitution

Moreover, (Γ_0, σ_0) can be found effectively from M.

Uniqueness of types theorem (Church version)

$\Gamma \vdash M : \sigma \;\&\; \Gamma \vdash M : \sigma' \Rightarrow \sigma = \sigma'$

Functional Programming language

ML is essentially $\lambda\!\to$Curry extended with

1. $\vdash Y : (\sigma \to \sigma) \to \sigma$

with reduction rule

$$Y \to_\delta \lambda f.f(Y\, f)$$

2. Types Nat, bool for the natural numbers and booleans with

$\vdash$ zero : Nat, $\vdash$ succ : Nat $\to$ Nat, $\vdash$ pred : Nat $\to$ Nat,

$\vdash$ zero? : Nat $\to$ bool

$\vdash$ conditional : bool $\to \sigma \to \sigma \to \sigma$ "if B then p else q"

pred (succ x) $\to_\delta$ x

zero? zero $\to_\delta$ true

zero? (succ x) $\to_\delta$ false

conditional true p q $\to_\delta$ p

conditional false p q $\to_\delta$ p

3. The let construction:

<u>let</u> id <u>be</u> $\lambda z.z$ <u>in</u> $\lambda fx.(id\ f)(id\ x)$

Intended meaning

$$\lambda fx.((\lambda z.z)\ f)((\lambda z.z)\ x)$$

$$(\lambda id\ \lambda fx.(id\ f)(id\ x))(\lambda z.z)$$

Theorem All computable functions can be represented in ML

Proof We only do primitive recursion. Let

$$f(0) = 13$$

$$f(k+1) = h(f(k),k)$$

and suppose that h is represented by H.
Then f can be represented by F such that

$$F\ k \twoheadrightarrow_\beta \text{if zero? } k \text{ then } 13 \text{ else } (H\ (F\ (\text{pred } k))\ (\text{pred } k))$$

Such an F can be found using Y. ■

There are two variants of functional languages, the eager and the lazy ones.

In an eager language evaluation of F A first A is reduced and then F acting on some normal form of M.

In a lazy language F A is evaluated directly and A is reduced later.

ML is usually eager. Clean and Haskell are lazy.

In lazy languages one can deal with “infinite” objects, like the list of all primes

[2,3,5,...]

and evaluate

take 3 [2,3,5,...] = [2,3,5]

Excerpts from a clean program

Projective view on a cube

```
implementation module matrix

import StdEnv

cons :: a [a] -> [a]
cons x xs = [x:xs]

zipWith :: (a->b->c) [a] [b] -> [c]
zipWith f xs ys = map (uncurry f) (zip2 xs ys)

repeat :: a -> [a]
repeat x = xs
              where xs = [x : xs]

:: Matrix :== [[Real]]

matmult :: Matrix Matrix -> Matrix
matmult xss yss = map f xss
                where
                 f a          = map (inprod a) (transpose yss)
                 inprod xs ys = sum (zipWith (*) xs ys)
                 transpose    = foldr (zipWith cons) (repeat [ ])
```

```
implementation module transformations

import StdEnv, matrix

// Points and linear maps

:: TwoDPoint        :==    (Real,Real)
:: ThreeDPoint      :==    (Real,Real,Real)
:: FourDPoint       :==    (Real,Real,Real,Real)

:: LinMap    :==    Matrix

// 4-D matrix application

toMatrix :: FourDPoint -> Matrix
toMatrix (x,y,z,w) =     [[x],[y],[z],[w]]

toPoint :: Matrix -> FourDPoint
toPoint [[x],[y],[z],[w]] = (x,y,z,w)

apply4 :: LinMap FourDPoint -> FourDPoint
apply4 m v = toPoint (matmult m (toMatrix v))

// projection of 4-D linear maps to 3-D maps via
//homogeneous coordinates

pointtohom :: ThreeDPoint -> FourDPoint
pointtohom (x,y,z) = (x,y,z,one)

homtopoint :: FourDPoint -> ThreeDPoint
homtopoint (x,y,z,w) = (x,y,z)

apply :: LinMap -> (ThreeDPoint -> ThreeDPoint)
apply f = homtopoint o (apply4 f) o pointtohom

// standard matrices

:: ThreeDVector :== (Real,Real,Real)
:: Angle  :== Real

rotationxmap :: Angle -> LinMap
rotationxmap t      =  [[one,   zero,   zero,     zero     ],
                        [zero,  cos t,  ~(sin t), zero     ],
                        [zero,  sin t,  cos t,    zero     ],
                        [zero,  zero,   zero,     one      ]]
```

Exercises

2.1 Solve

(i) $\vdash W : ?\quad W = \lambda xy.xyy$

(ii) $\vdash ? : (\sigma \to \tau) \to (\tau \to \rho) \to (\sigma \to \rho)$

(iii) $? \vdash f\,[x,y] : p$

2.2 Prove that of the following two terms one is typable in $\lambda\!\to$ and the other not.

$\lambda xy.x(xI)y \qquad \lambda xy.x(Ix)y$

2.3 Write a functional program for the factorial.

2.4 Prove that the normalization theorem implies that not all computable functions are representable in $\lambda\!\to$.

2.5 A type σ is called inhabited if $\vdash M : \sigma$ for some term M. Prove the following result by Statman. Let

$$\sigma = \sigma_1 \to \sigma_2 \to \ldots \to \sigma_n \to p$$

be a type containing only p as type variable. Then

$$\sigma \text{ is inhabited } \Leftrightarrow \exists i \in \{1,\ldots,n\}\ \sigma_i \text{ is not inhabited.}$$

This gives a decision method for inhabitation for types built from only one type variable.

3 Interactive functional programs

Autistic programming

Compute π in 100 decimals

Pi 100 $\twoheadrightarrow_\beta$ 3.141592653589....

Interactive programming

Most contemporary applications

Control of traffic, factory, system

Simple example:

read two numbers and print their difference

ML

P = write (read - read)

Continuations

λ + read + write + stop

Semantics

$$\frac{M}{M^{hnf}}$$

$$\frac{\text{read } F}{F\ a_1} \qquad \frac{\text{write } b\ F}{F} \qquad \frac{\text{stop}}{}$$

Input stream	Output stream
$a_1, a_2, \ldots.$	$b, \ldots$

The process of reading two inputs and printing the sum becomes

$$P \equiv \text{read}(\lambda x.\text{read}(\lambda y.\text{write } (x+y) \text{ stop}))$$

The process of continuously reading two inputs and printing the sum becomes

$$Q \equiv \text{read}(\lambda x.\text{read}(\lambda y.\text{write } (x+y) \; Q))$$

$$\equiv Y(\lambda q.\text{read}(\lambda x.\text{read}(\lambda y.\text{write } (x+y) \; q)))$$

This essentially happens in the language Haskell

Using this idea arbitrary interactive programs can be written

Disadvantages

- This interaction is obtained by 'delegation'

Some of the output b's have to be interpreted

Put 7+7 on the screen:

write 'echo (7+7)' stop

Print 7+7:

write 'lpr (7+7)' stop

- Execution order has to be overspecified

Print 7+7 and put it on the screen (any order)

write 'echo (7+7)' (write 'lpr (7+7)' stop)

write 'lpr (7+7)' (write 'echo (7+7)' stop)

There is a natural solution without having to rely on 'theoretical' non-determinism.

The world as values

In the language Clean interaction is not done via delegation, but by direct operation.

In order to describe the method we want to be more explicit about what happens with continuations.

Let

$$\text{In} \quad = [a_1, a_2, a_3, \ldots]$$
$$\text{Out} \quad = [\ldots , b_3, b_2, b_1]$$

be the input and output streams. We want to mention them explicitly with the continuations.

$$\text{read } F \langle [a,\text{In}], \text{Out}\rangle = F\ a \langle \text{In}, \text{Out}\rangle$$

$$\text{write } b\ F \langle \text{In}, \text{Out}\rangle = F \langle \text{In}, [b, \text{Out}]\rangle$$

In this way the input and output streams are taken into the functional world and operated on.

One has to be careful: In and Out may not be copied, the have to be unique. This can be checked by a type system.

Having that, the umbellical cord to the world

<In, Out>

can be improved by having as copy of the world something like

<keybord, mouse, screen, files, printer>

Uniqueness types

Want a type system such that

$f : \mathrm{File}, \mathrm{write} : \mathrm{File} \to \mathrm{Char} \to \mathrm{File} \vdash \ldots \mathrm{write}\ \text{'a'}\ f \ldots$

warrants that f occurs only one time at the RHS

$\lambda\!\to$ resource conscious version

$x : \sigma \vdash x : \sigma$ $\Gamma, x : \sigma \vdash M : \tau \Rightarrow \Gamma \vdash (\lambda x.M) : (\sigma \to \tau)$ $\Gamma \vdash F : (\sigma \to \tau)\ \&\ \Delta \vdash a : \sigma\ \&\ \Gamma, \Delta$ disjoint $\Rightarrow \Gamma, \Delta \vdash (F\ a) : \tau$
$\Gamma \vdash M : \tau \Rightarrow \Gamma, x : \sigma \vdash M : \tau$ weakening
$\Gamma, x : \sigma, x' : \sigma \vdash M : \tau$ $\Rightarrow \Gamma, y : \sigma \vdash M[x{:=}y, x'{:=}y] : \tau$ contraction

$\lambda\!\to\!L$ first three rules

$\lambda\!\to\!A = \lambda\!\to\!L$ + weakening

$\lambda\!\to\ = \lambda\!\to\!A$ + contraction

$\lambda{\to}U$ (Barendsen & Smetsers) is the following system

Type annotations

type = typevar | atype $\to$ atype

atype = type | * type

Subtyping

$*p \le p$

$A{\to}B \le A'{\to}B' \Leftrightarrow *(A{\to}B) \le *(A'{\to}B') \Leftrightarrow A' \le A \;\&\; B \le B'$

Permissiveness [ ] : atype $\to$ type

$[p] = [*p] = p$

$[A{\to}B] = A \to B$

$[*(A{\to}B)] = \uparrow$ (undefined)

$x : \sigma \vdash x : \sigma$

$\Gamma, x : \sigma \vdash M : \tau \Rightarrow \Gamma \vdash (\lambda x.M) : {}^{\cap\Gamma}(\sigma{\to}\tau)$

where $\cap\Gamma = *$ if $(z{:}{*A}) \in \Gamma$, nothing else

$\Gamma \vdash F : (\sigma \to \tau) \;\&\; \Delta \vdash a : \sigma \;\&\; \Gamma,\Delta$ disjoint $\Rightarrow \Gamma, \Delta \vdash (F\ a) : \tau$

$\Gamma \vdash M : \tau \Rightarrow \Gamma, x : \sigma \vdash M : \tau$ weakening

$\Gamma \vdash M : \sigma,\ \sigma \le \tau \Rightarrow \Gamma \vdash M : \tau$ subsumption

$\Gamma, x : [\sigma], x' : [\sigma] \vdash M : \tau$

$\Rightarrow \Gamma, y : \sigma \vdash M[x{:=}y, x'{:=}y] : \tau$ [ ]-contraction

Example

```
implementation module figureio

import StdEnv
import deltaEventIO, deltaIOSystem, deltaPicture, deltaWindow


IOStart :: *s (Int,Int) (Keybdfct *s (IOState *s)) (UpdateFunction *s) *World
-> *World
IOStart initstate windowsize keybdfct updatefunction world = CloseEvents
events` world`
where
   (s, events`)          =     StartIO [menu, window] initstate [ ] events
   (events, world`)      =     OpenEvents world

   menu          =     MenuSystem [file];

   file          =     PullDownMenu 1 "File" Able
                       [ MenuItem  2 "Quit" (Key 'Q') Able Quit]


   window     =    WindowSystem [ ScrollWindow 3 (0,0) "Picture"
                   (ScrollBar (Thumb 0) (Scroll 10)) (ScrollBar (Thumb 0)
                   (Scroll  10))((0,0), (1000,1000)) (50,50)
                   windowsize  updatefunction
                   [Keyboard Able keybdfct, GoAway Quit]]

Quit state io =  (state, QuitIO io)


Start :: * World -> * World
Start world  =
      IOStart  InitState  (windowwidth,windowheight)  KeyboardHandler
Update  world
```

Things go well because menu operations are higher-order functions and these can be handled

For Clean information, a quality compiler and examples can be obtained from

http://www.cs.kun.nl/~clean

3. Exercises

3.1 Write a continuation program that reads from the input list of integers and adds them until a zero appears; then the sum obtained thus far is put on the output stream and the process is stopped.

3.2 Write a continuation program that reads from the input list of integers and puts the square of each non-zero integer on the output stream until a zero comes in; then the input is discarded and the process waits until the next zero comes in; then the process continues putting the squares of the (non-zero) numbers on the output stream; etcetera forever.

4 The quest for correctness

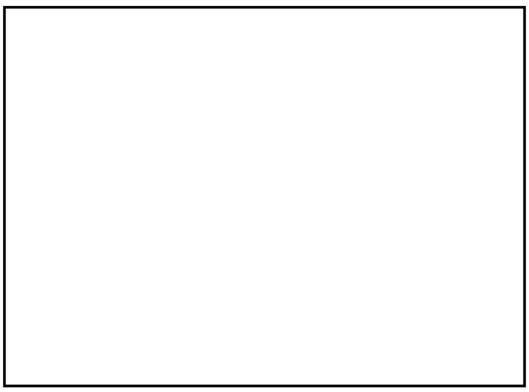
Correctness: becomming commercially important

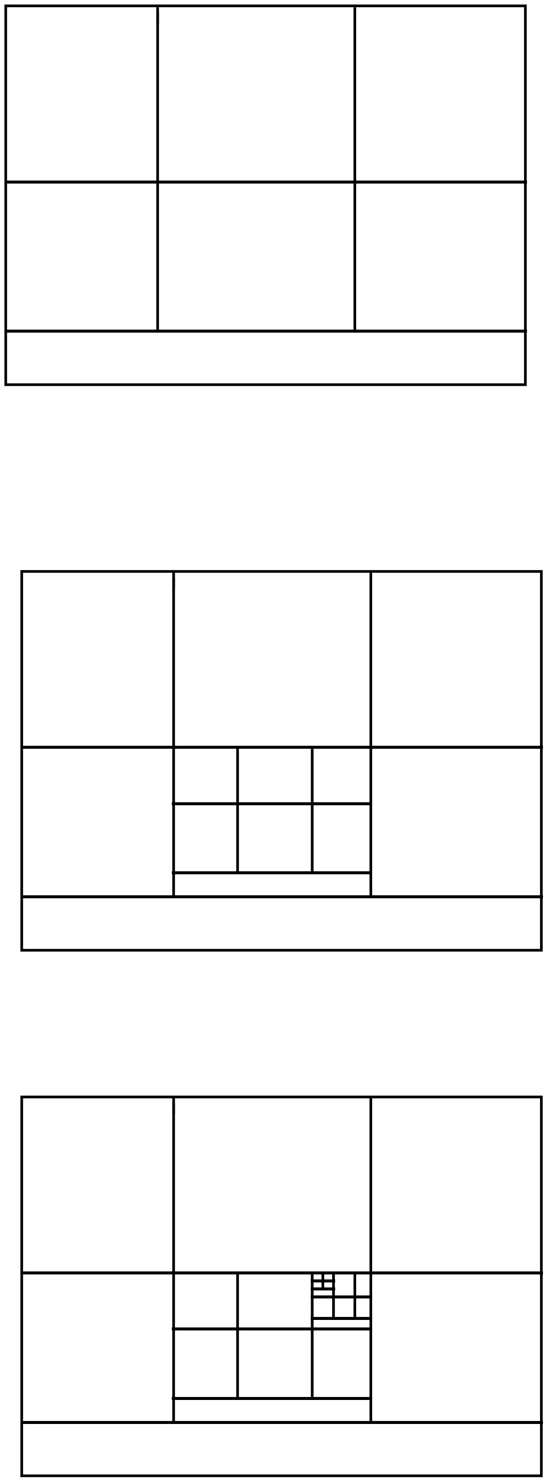
scientifically this was always the case

Technology

Products consisting of components

consisting of components ....





The Chinese box

Compositional modules

$$S_1(x_1), \ldots , S_n(x_n) \vdash S(x)$$

where $x = f(x_1, \ldots , x_n)$

For reliable products we want proofs here

Hardware	$\approx$	propositional logic
Software	$\approx$	predicate logic
Mathematics	$\approx$	predicate logic + computations

Proofs of understandable statements are important

But may be difficult

Why are they correct?

- Understanding by anybody
- Understanding by trained person
- Sociological verification: peer reviews
- Machine verification of formal versions

Aim: highest degree of certainty

Should we believe machine checked proofs?

Methodology (N.G. de Bruijn)

The verifying program should be small

small enough to be checked by hand

Case study: proof-checking mathematics

- understandable statements
- non-trivial
- will have spin-off for verification of programs

Notion of proof in mathematics

Thales ± 600 BC	first proofs
Plato ± 400 BC	emphasis on importance of proofs
Aristotle ± 300 BC	axiomatic method
	quest for logic
	proof verification ≠ proof finding
Euclid ± 275 BC	axiomatic geometry
Frege ± 1870	full description of logic
Russell ± 1910	formalised mathematics
de Bruijn ± 1970	computer verification in type theory

In mathematics

In context Γ we have A

Logic

$\Gamma \vdash_L A$ because of proof p

Type theory

$[\Gamma] \vdash_\lambda [p] : [A]$

Automated verification

$\text{type}_{[\Gamma]}([p]) = [A]$

Statement A of predicate logic are translated as types

Curry, Howard, de Bruijn:

propositions—as—types interpretation

$[A]$	=	type (set) of proofs of A
$[A \supset B]$	=	$[A] \to [B]$
$[\forall x \in X.P]$	=	$\Pi x{:}X.[P]$

Example

$\Gamma = X{:}\ \text{set},\ P{:}X \to \text{prop}$

$\Gamma \vdash_{\lambda} (\lambda y{:}[Px].y) : [Px \supset Px]$

$\Gamma \vdash_{\lambda} (\lambda x{:}X\ \lambda y{:}[Px].y) : [\forall x{:}X.Px \supset Px]$

Other example

Proposition. Let R be a binary relation on a set A. Then

$$R \text{ is antisymmetric} \rightarrow R \text{ is irreflexive}.$$

Proof. Antisymmetry is

$$\forall ab[Rab \rightarrow \neg Rba].$$

Let $a \in A$ be arbitrary and suppose

$$Raa.$$

Then

$$\neg Raa,$$

contradiction. Therefore

$$\forall a\, \neg Raa. \blacksquare$$

In lambda notation.

Γ = A:set, R:A→A→prop

$\Gamma \vdash$?? : antisym R → irrefl R.

?? = λp:antisym R λa:A λq:irrefl R.paaqq.

Indeed

Γ, p : antisym R	$\vdash$ p : ∀ab[Rab → Rba → ⊥]
Γ, p : antisym R, a:A	$\vdash$ paa : Raa → Raa → ⊥
Γ, p:antisym R, a:A, q:Raa	$\vdash$ paaqq : ⊥
Γ, p : antisym R, a:A	$\vdash$ λq:Raa.paaqq : Raa → Raa →⊥
Γ, p : antisym R	$\vdash$ λa:A λq:Raa.paaqq : ∀a [Raa → Raa →⊥] = irrefl R

$\Gamma \vdash$ <u>λp:antisym R λa:A λq:Raa.paaqq</u> :

antisym R → irrefl R

Curry version: λpaq.paaqq

Hilbert style proof of $\quad$ antisym R $\vdash$ irrefl R

Assume

$$\text{antisym R} \equiv \forall ab\ [Rab \rightarrow Rba \rightarrow \bot]$$

so

$$Raa \rightarrow Raa \rightarrow \bot$$

We know

$$(Raa \rightarrow Raa \rightarrow \bot) \rightarrow (Raa \rightarrow \bot) \qquad (*)$$

so

$$Raa \rightarrow \bot \equiv \text{irrefl R}$$

As to (*)

assume	$p \rightarrow p \rightarrow \bot$
ax	$(p \rightarrow (q \rightarrow r)) \rightarrow (p \rightarrow q) \rightarrow (p \rightarrow r)$
subst	$(p \rightarrow ((r \rightarrow p) \rightarrow p)) \rightarrow (p \rightarrow (r \rightarrow p)) \rightarrow (p \rightarrow p)$
ax	$p \rightarrow (q \rightarrow p)$
subst	$p \rightarrow ((r \rightarrow p) \rightarrow p)$
MP	$(p \rightarrow (r \rightarrow p)) \rightarrow (p \rightarrow p)$
ax	$(p \rightarrow (r \rightarrow p))$
MP	$(p \rightarrow p)$
ax	$(p \rightarrow (p \rightarrow \bot)) \rightarrow (p \rightarrow p) \rightarrow (p \rightarrow \bot)$
MP	$(p \rightarrow p) \rightarrow (p \rightarrow \bot)$
MP	$p \rightarrow \bot$

Natural deduction proofs $\approx$ lambda terms
Hilbert style proofs $\approx$ combinators

$$\mathrm{Id} = \lambda x.x = S\,K\,K$$

with

$$S = \lambda xyz.xz(yz)$$

$$K = \lambda xy.x$$

Translation

$$\lambda xy.yx = S\,(K\,(S\,I))\,K$$

Translation from λ-term into combinatory term is exponential or if one uses suitably chosen combinators quadratic. Best result by

Statman

$$O(n.\log n)$$

Conclusion

Better use lambda terms

Constructing formal proofs

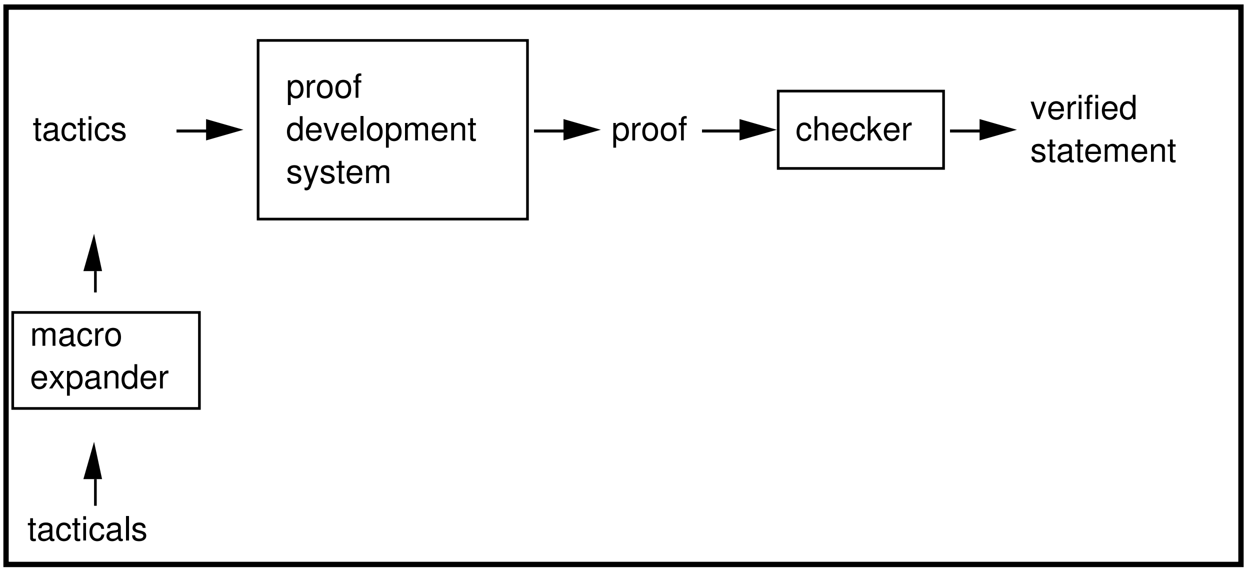
Question How do we obtain proof-objects?

Proofs can be produced by

- a trained person
- <u>a cooperation between a trained person and a computer</u>

Interactive proof development systems

Lego, Coq



Goal

Producing proof-objects with the same effort as writing in, say, LaTeX

Pure Type Systems

General rules PTS

Start

$$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad x \text{ fresh}$$

Weakening

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}\ C \vdash \ : B} \qquad x \text{ fresh}$$

Application

$$\frac{\Gamma \vdash F : (\Pi x{:}A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$$

Abstraction

$$\frac{\Gamma, x{:}A \vdash A : B \quad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A) : (\Pi x{:}A.B)}$$

Conversion

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \qquad B =_\beta B'$$

term ::= var | const | term term | λ var:term term | Π var:term term
context ::= $<x_1 : A_1, \ldots , x_n : A_n>$
statement ::= term : term

Specific axioms and rules for PTS

Specification of a PTS

Sorts	S	$s_1, s_2, \ldots.$
Axioms	A	$s_1 : s_2, \ldots.$
Rules	R	$(s_1, s_2, s_3), \ldots.$

Let $s_1, s_2, s_3 \in S$. The following rules are declared by the specification of the PTS

axioms $\quad \langle\rangle \vdash s_1 : s_2 \quad$ for $s_1 : s_2$ in A

Product

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3} \quad \text{for } (s_1, s_2, s_3) \text{ in } R$$

Write $(s_1, s_2) = (s_1, s_2, s_2)$

$\lambda\!\rightarrow$ Simply typed lambda calculus

Propositional logic

S	$*, \Box$
A	$* : \Box$
R	$(*, *)$

$\lambda 2$ Second order lambda calculus

Second order propositional logic

S	$*, \Box$
A	$* : \Box$
R	$(*, *), (\Box, *)$

λP Dependent types

S	$*, \Box$
A	$* : \Box$
R	$(*, *), (\Box, *)$

λC Calculus of constructions

S	$*, \Box$
A	$* : \Box$
R	$(*, *), (\Box, *), (*, \Box), (\Box, \Box)$

Information about <u>proof assistents</u> by Frank Pfenning, CMU, under the name "Logical Frameworks" can be obtained from

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/fp/www/lfs.html

or via my home page

http://www.cs.kun.nl/~henk/

Exercises

4.1 Construct a lambda term p such that

$X{:}set, P{:}X \to prop, Q{:}prop \vdash p :$
$\forall x.(Px \to Px \to Q) \to Px \to Q$

Hence p is a proof-object for
$\forall x.(Px \to Px \to Q) \to Px \to Q$.

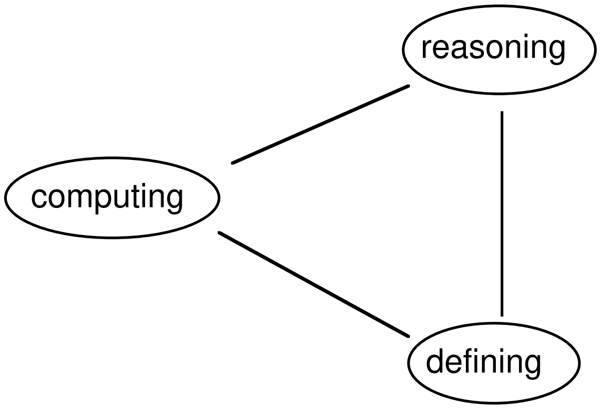
We can do this in λP taking set = prop = *.

4.2 Construct a proof in $\lambda 2$ of

$(\forall p{:}* \, (A \to B \to p) \to p) \to B$.

5 **Computations and proofs**

Doing mathematics



Computations are needed for asserting e.g. the following statements

$[\sqrt{45}] = 6$

Prime (61)

$(x+1)(x-1) = x^2 - 1$

Babylonians were good at computations but no proofs

Greek were good at proving but had few computations

Formal proofs of computations should not be done in first order predicate logic with equality

Law of Ruys:

proofs of an equation are quadratic in size of statement

Poincaré principle

If in a mathematical argument we need

$$2 + 2 = 4$$

this is not a proof in the strict sense, but just a verification

In type systems this becomes

$$\left.\begin{array}{l} p \text{ proves } A(t) \\ t \to_R s \end{array}\right\} \quad \Rightarrow \quad p \text{ proves } A(s)$$

de Bruijn adopted the PP for $\beta\delta$-reduction

Scott and Martin-Löf later for ι-reduction

Recursor R for primitive recursion over natural numbers but also trees and other data structures

$$R\ a\ b\ \underline{0} \to_\iota a$$

$$R\ a\ b\ \underline{n+1} \to_\iota b\ \underline{n}\ (R\ a\ b\ \underline{n})$$

Examples

Using R one can make an F such that

$$F\ \underline{n} \rightarrow_{\beta\delta\iota} [\sqrt{n}]$$

Proof obligation

$$\forall n\ (F\ n)^2 \leq n < ((F\ n)+1)^2$$

Symbolic computing consists of manipulations with syntactic expressions

$$x+1 : \mathrm{Int}$$

$$`x+1' : \mathrm{term}(\mathrm{Int})$$

There is a self-interpreter

$$E\ `t' \to_{\beta\delta\iota} t$$

There is a term simplify such that

$$\mathrm{simplify}\ `(x+1)(x-1)' \to_{\beta\delta\iota} `x^2 - 1'$$

Proof obligation

$$\forall\ t{:}\mathrm{term}(\mathrm{Int}).\ E(\mathrm{simplify}\ t) = E\ t$$

Then

$$E(\text{simplify } `(x+1)(x-1)') = E\ `(x+1)(x-1)'$$

$$E\ `x^2 - 1' \qquad\qquad (x+1)(x-1)$$

$$x^2 - 1$$

so

$$x^2 - 1 = (x+1)(x-1)$$

Goes smoothly in new versions of Lego and Coq

For checking primality, one can construct from the recursor R a function K_{Prime} such that

$$
\begin{aligned}
K_{Prime}\ \underline{n} &= \underline{true} && \text{if n is a prime;}\\
&= \underline{false} && \text{else.}
\end{aligned}
$$

Proof obligation

$$\forall n\ [(\text{Prime } n \Leftrightarrow K_{Prime}\ n = \underline{true})$$
$$\&\ (K_{Prime}\ n = \underline{true} \text{ or } K_{Prime}\ n = \underline{false})]$$

where

$$\text{Prime } n \Leftrightarrow \forall d{<}n\ (d \mid n \rightarrow d = \underline{1})\ \&\ n{>}1$$

M. Oostdijk automatised this for all primitive recursive functions and predicates

H. Elbers constructed by hand a different K_{Prime} by applying Fermat's little theorem
(together with the needed proof of correctness).

General pattern of computations

$$p \to_1 \; \dots \; \to_1 p^{1\text{-nf}} = f_1(p)$$

$$p \to_2 \; \dots \; \to_2 p^{2\text{-nf}} = f_2(p)$$

....

....

$$F_1\, p \to_{\beta\delta\iota} \; \dots \; \to_{\beta\delta\iota} f_1(p)$$

$$F_2\, p \to_{\beta\delta\iota} \; \dots \; \to_{\beta\delta\iota} f_2(p)$$

....

....

with proof obligations

$$\forall p\; S_1(p, F_1\, p)$$

$$\forall p\; S_2(p, F_2\, p)$$

....

....

Extending the use of the Poincaré Principle

Fixedpoint reduction

$$Y\, f \;\to_Y\; f\,(Y\, f)$$

Arithmetic

$$\text{add}\; \underline{n}\; \underline{m} \;\to_A\; \underline{n+m}$$

Conclusion

Computer Algebra

- Representing $\sqrt{2}$ exactly
- Symbolic computations

Computer Mathematics

- Representing exactly

  $X = \{n \in N \mid \neg \exists x_1 \ldots x_k \; p(x_1, \ldots, x_k, n) = 0\}$
- Stating properties about infinity

We can state with confidence that

$$3 \in \{n \in N \mid \neg \exists x_1 \ldots x_k \; p(x_1, \ldots, x_k, n) = 0\}$$

or

There are infinitely many primes

because of having proofs

Even if a statement A may not be decidable, the statement

p proves A

is decidable

Applications of Computer Mathematics

- Different function of referees
- Library of Mathematics
- Education

  Interactive books
- Interactive theorem proving
- Computational meaning of theorems

  $\vdash \forall x \exists y A(x,y) \quad \Rightarrow$

  $\exists$ f computable $\vdash A(x,f(x))$

  provided that A is decidable
- Numerical values automatically

Tactics

```
Goal {x:nat} Ex [y:nat] and (less_nat x y) (is_prime y);
  Intros x;
   z == succ (fac x);            (* let z be x! + 1*)
   Refine has_prime_factor z;    (* we have a prime factor of z, if 1 < z   *)
    Refine le2less;              (* we have 1 < z, if 1 <= x!    *)
     Refine faculty_lemma;       (* prove 1 <= x!        *)
  Intros y PF;                   (* assume y, assume prime factor(y,z)*)
   D == fst PF : divides y z;    (* so y | z *)
   P == snd PF : is_prime y;     (* so prime(y)*)
   H == fst P  : less one y;     (* so 1 < y      *)
  Refine ExIntro |? ?| y;        (* take y and prove x < y & is_prime(y) *)
  Refine pair ? P;               (* we have x < y and is_prime(y), if x < y*)
   Refine less2not_le;           (* we have x < y, if not(y <= x)  *)
  Intros H1;                     (* assume y <=x, prove absurd      *)
   Refine less_irrefl one;       (* we have absurd, if 1 < 1     *)
  Refine less_exten ? ? H;       (* we have 1 < 1, if 1=1 & y = 1 & 1 < y *)
   Refine eq_refl;               (* prove 1 = 1*)
  Refine divides_lemma ? D;      (* we have y = 1, if y|x! & y|z *)
  Refine fac_divides ? ? ? H1;   (* we have y|x!, if 1 <= y & y <= x *)
   Refine less2le;               (* we have 1 <= y, if 1 < y + 1 *)
  Refine less_succ H;            (* prove 1 < y + 1, using 1 < y  *)
```

Proof-object (M. Ruys)

```
= [x:el Nat]infinitely_bounded_primes_exist x (Ex%%(el Nat) ([y:el
Nat]and (ap2%%Nat%%Nat%%Omega LessN x y) (is_prime y))) ([y:el
Nat][H:and (and (ap2%%Nat%%Nat%%Omega LessN x y)
(ap2%%Nat%%Nat%%Omega LessEqN y (succ (fac x)))) (is_prime
y)]ExIntro%%(el Nat) y ([y'4:el Nat]and (ap2%%Nat%%Nat%%Omega
LessN x y'4) (is_prime y'4)) (pair%%(ap2%%Nat%%Nat%%Omega
LessN x y)%%(is_prime y) (fst%%(ap2%%Nat%%Nat%%Omega LessN x
y)%%(ap2%%Nat%%Nat%%Omega LessEqN y (succ (fac x))) (fst%%(and
(ap2%%Nat%%Nat%%Omega LessN x y) (ap2%%Nat%%Nat%%Omega
LessEqN y (succ (fac x))))%%(is_prime y) H)) (snd%%(and
(ap2%%Nat%%Nat%%Omega LessN x y) (ap2%%Nat%%Nat%%Omega
LessEqN y (succ (fac x))))%%(is_prime y) H)))];
```

Proposed systems for Computer Mathematics

TS = PTS + extra reduction

Make a general system with a ‘joystick’