Mechanized program verification and interactive development

Theoretical foundations of proof assistants and applications

Marcus Vinícius Midena Ramos

UFPE/UNIVASF

February 06&07, 2014

mvmr@cin.ufpe.br
marcus.ramos@univasf.edu.br
(7 de julho de 2015, 17:45)

Who am I?

- Electronics Engineering at Universidade de São Paulo in 1982;
- Master in Digital Systems at Universidade de São Paulo in 1991;
- Teaching experience with programming languages, compilers, formal languages, automata theory and computation theory since 1991;
- Current position at Universidade Federal do Vale do São Francisco (Petrolina-PE/Juazeiro-BA) since 2008;
- PhD in Computer Science student at Universidade Federal de Pernambuco since 2011.

Why I am here?

- Working with formal mathematics and proof assistants;
- Research on language and automata theory formalization;
- Invitation from the chair;
- Bring new ideas, share the experience and motivate new users;
- Suggest a framework for adaptive technology theory development.

What is this all about?

- Formal mathematics;
- Interactive theorem proving;
- Interactive program development;
- Proof assistants;
- Coq.

Objectives

- Introduce Interactive Proof Assistants;
- Discuss their role in program development and theorem proving;
- Present some important formalization projects (academic and industrial);
- Present highlights of the underlying formal theory;
- Introduce the Coq proof assistant;
- Show some examples;
- Present my research area along with some results.

Summary

How we make it

- Today: from 1 to 8
  - Theory;
  - Mathematics;
  - "Hard" part.
- Tomorrow: from 9 to 13
  - General information;
  - Examples and cases;
  - "Easy" part.

History and current practice

- Theorem proofs:
  - Informal;
  - Difficult to build;
  - Difficult to check.
- Computer programs:
  - Informal;
  - Difficult to build;
  - Difficult to test.
- Coincidence?

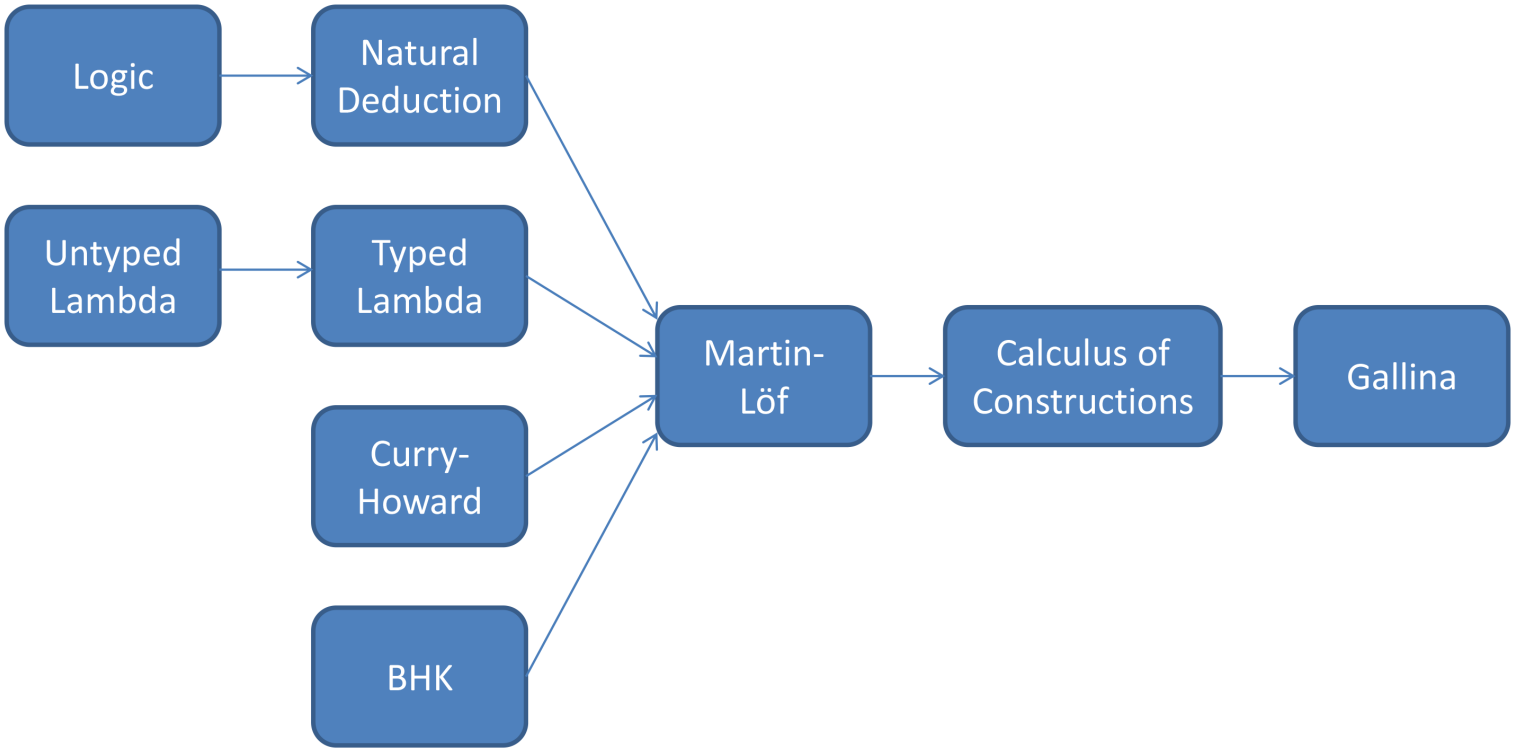
History and current practice

- **NOT REALLY**, as theorem proving and software development have essentially the same nature;
- According to the Curry-Howard Isomorphism, to develop a program is the same as to prove a theorem, and vice-versa;
- Exploring this similarity his can be benefitial to both activities:
  - Reasoning can be brought into programming, and
  - Computational ideas can be used in theorem proving.
- How to improve both then?

Perspectives

- Formalization (*"computer encoded mathematics"*) is the answer;
- Computer-aided reasoning;
- Use of proof assistants, also known as interactive theorem provers.

Formal mathematics



Propositional Logic

- Formulas that use *propositional variables* and *logical connectives*.

$$
\begin{array}{rcl}
formula & ::= & variable \\
& | & \bot \\
& | & \top \\
& | & (formula \wedge formula) \\
& | & (formula \vee formula) \\
& | & (formula \Rightarrow formula) \\
& | & (formula \Leftrightarrow formula) \\
& | & (\neg formula) \\
variable & ::= & a \,|\, b \,|\, c \,|\, ...
\end{array}
$$

Proposicional Logic

Logical connectives:

- $\wedge$: Conjunction ("and");
- $\vee$: Disjunction ("or");
- $\Rightarrow$: Implication ("if-then");
- $\Leftrightarrow$: Bi-implication ($(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$) ("if-and-only-if");
- $\neg$: Negation ($\neg a \equiv a \Rightarrow \bot$) ("not");
- $\bot$: False;
- $\top$: True ($\top \equiv \bot \Rightarrow \bot$).

Predicate Logic

- Propositional formulas with the addition of *quantifiers* and *predicates*.

$$
\begin{array}{rcl}
formula & ::= & variable \\
& | & \bot \\
& | & \top \\
& | & pred\_name\,(arg\_list) \\
& | & (formula \wedge formula) \\
& | & (formula \vee formula) \\
& | & (formula \Rightarrow formula) \\
& | & (formula \Leftrightarrow formula) \\
& | & (\neg formula) \\
& | & (\forall\, variable\,.\, formula) \\
& | & (\exists\, variable\,.\, formula)
\end{array}
$$

Predicate Logic

$$
\begin{array}{rcl}
variable & ::= & a \,|\, b \,|\, c \,|\, ... \\
pred\_name & ::= & P_0 \,|\, P_1 \,|\, P_2 \,|\, ... \\
arg\_list & ::= & term \,|\, arg\_list\,,\, term \\
term & ::= & fun\_name\,(arg\_list) \,|\, term\_var \,|\, term\_const \\
term\_var & ::= & v_0 \,|\, v_1 \,|\, v_2 \,|\, ... \\
term\_const & ::= & c_0 \,|\, c_1 \,|\, c_2 \,|\, ...
\end{array}
$$

Predicate Logic

Logical quantifiers:

- $\forall$: Universal quantifier ("for all");
- $\exists$: Existential quantifier ("exists").

Examples

1. $(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$
2. $(a \wedge b) \Rightarrow (b \wedge a)$
3. $(a \vee (a \wedge b)) \Rightarrow a$
4. $(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)$
5. $\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$

Characteristics

- Calculus for theorem proving;
- Part of Proof Theory;
- Based in simple inference rules that resemble the rules of natural thinking;
- Each connective is associated to introduction and elimination rules;
- The proof of a theorem (proposition) is a structured sequence of inference rules that validate the conclusion, usually without depending on any hypothesis;
- The proof is represented as a tree;
- Gentzen (1935) and Prawitz (1965);
- Originally developed for propositional logic, was later extended for predicate logic.

Inference rules for implication ($\Rightarrow$)

Introduction:

$$\frac{\begin{array}{c}[a]\\ \dots \\ b\end{array}}{a \Rightarrow b}\ (\Rightarrow I)$$

Elimination:

$$\frac{a \Rightarrow b \qquad a}{b}\ (\Rightarrow E)$$

Example 1: proof tree

Theorem:

$$(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$$

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow (b \Rightarrow c) \qquad a}{b \Rightarrow c}\,(\Rightarrow E) \qquad b}{c}\,(\Rightarrow E)}{a \Rightarrow c}\,(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

Inference rules for conjunction ($\wedge$)

Introduction:

$$\frac{a \qquad b}{a \wedge b}\ (\wedge I)$$

Elimination 1:

$$\frac{a \wedge b}{a}\ (\wedge E_1)$$

Elimination 2:

$$\frac{a \wedge b}{b}\ (\wedge E_2)$$

Example 2: proof tree

Theorem:

$$(a \wedge b) \Rightarrow (b \wedge a)$$

Proof:

$$\dfrac{\dfrac{\dfrac{a \wedge b}{b}\,(\wedge E) \qquad \dfrac{a \wedge b}{a}\,(\wedge E)}{b \wedge a}\,(\wedge I)}{(a \wedge b) \Rightarrow (b \wedge a)}\,(\Rightarrow I)$$

Inference rules for disjunction ($\vee$)

Introduction 1:

$$\frac{a}{a \vee b}\ (\vee I_1)$$

Introduction 2:

$$\frac{b}{a \vee b}\ (\vee I_2)$$

Elimination:

$$\frac{\begin{array}{ccc} & \begin{array}{c}[a]\\ \vdots\\ c\end{array} & \begin{array}{c}[b]\\ \vdots\\ c\end{array} \\ a \vee b & & \end{array}}{c}\ (\vee E)$$

Example 3: proof tree

Theorem:

$$(a \vee (a \wedge b)) \Rightarrow a$$

Proof:

$$\dfrac{\dfrac{a \vee (a \wedge b) \qquad \begin{matrix}[a] \\ a\end{matrix} \qquad \dfrac{[a \wedge b]}{a}\,(\wedge E)}{a}\,(\vee E)}{(a \vee (a \wedge b)) \Rightarrow a}\,(\Rightarrow I)$$

Inference rules for false ($\bot$)

Introduction:

No rule.

Elimination (*ex falso quodlibet*):

$$\frac{\bot}{a}\ (\bot E)$$

Inference rules for negation ($\neg$)

Introduction (same as implication introduction):

$$\frac{\begin{array}{c}[a] \\ \dots \\ \bot\end{array}}{\neg a}\ (\neg I, \text{ same as } \Rightarrow I)$$

Elimination (same as implication elimination):

$$\frac{a \qquad \neg a}{\bot}\ (\neg E, \text{ same as } \Rightarrow E)$$

Inference rules for negation ($\neg$)

"Elimination" (*reduction ad absurdum*):

$$\begin{array}{c} [\neg a] \\ \dots \\ \dfrac{\bot}{a}\,(RAA) \end{array}$$

Classical logic only.

Example 4: proof tree

Theorem:

$$(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)$$

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow b \qquad a}{b}\,(\Rightarrow E) \qquad \neg b}{\bot}\,(\neg E)}{\neg a}\,(\Rightarrow I)}{\neg b \Rightarrow \neg a}\,(\Rightarrow I)}{(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)}\,(\Rightarrow I)$$

Inference rules for universal quantifier ($\forall$)

Introduction:

$$\frac{A(x)}{\forall x.A(x)}\ (\forall I)$$

Elimination:

$$\frac{\forall x.A(x)}{A[t/x]}\ (\forall E)$$

Inference rules for existential quantifier ($\exists$)

Introduction:

$$\frac{A[t/x]}{\exists x.A(x)}\ (\exists I)$$

Elimination:

$$\frac{\exists x.A(x) \qquad \begin{array}{c}[A[t/x]] \\ \vdots \\ B\end{array}}{B}\ (\exists E)$$

(B cannot have free variables introduced by A)

Example 5: proof tree

Theorem:

$$\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$$

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x.R(x,x)}{R(x,x)}\,(\forall E)}{\exists y.R(x,y)}\,(\exists I)}{\forall x.\exists y.R(x,y)}\,(\forall I)}{(\forall x.R(x,x) \Rightarrow (\forall x.\exists y.R(x,y))}\,(\Rightarrow I)$$

Características

Sistema formal para representação de computações.

- Baseado na definição e aplicação de funções;
- Funções são tratadas como objetos de ordem mais elevada, podendo ser passados como argumentos e retornados de outras funções;
- Simplicidade: possui apenas dois comandos;
- Permite a combinação de operadores e funções básicas na geração de operadores mais complexos;
- Mesmo na versão *pura* (sem constantes), permite a representação de uma ampla gama de operações e tipos de dados, entre números inteiros e variáveis lógicas;
- Versões não-tipada e tipada.

História

- Alonzo Church, 1903-1995, Estados Unidos;
- Inventou o Cálculo Lambda na década de 1930;
- Resultado das suas investigações acerca dos fundamentos da matemática;
- Pretendia formalizar a matemática através da noção de funções ao invés da teoria de conjuntos;
- Apesar de não conseguir sucesso, seu trabalho teve grande impacto em outras áreas, especialmente na computação.

Aplicações

Modelo matemático para:

- Teoria, especificação e implementação de linguagens de programação baseadas em funções, especialmente as linguagens funcionais;
- Verificação de programas;
- Representação de funções computáveis;
- Estudo da computabilidade;
- Teoria das provas.

Foi usado na demonstração da indecidibilidade de diversos problemas da matemática, antes mesmo dos formalismos baseados em máquinas.

Motivação

Considere a expressão $x - y$. Ela pode ser formalizada, na notação matemática usual, através de funções com um único parâmetro:

- $f(x) = x - y$, ou
- $g(y) = x - y$.

ou, ainda:

- $f : x \mapsto x - y$, ou
- $g : y \mapsto x - y$.

Por exemplo:

- $f(0) = 0 - y$, ou
- $f(1) = 1 - y$.

Motivação

Representação dessas funções na linguagem lambda:

- $f = \lambda x.x - y$, ou
- $g = \lambda y.x - y$.

A aplicação da função a um argumento é representada pela justaposição da função ao argumento:

- $(\lambda x.x - y)(0) = 0 - y$, ou
- $(\lambda x.x - y)(1) = 1 - y$.

Motivação

Funções com múltiplos parâmetros:

- $h(x, y) = x - y$, ou
- $k(y, x) = x - y$.

Podem ser representadas na linguagem lambda como:

- $h = \lambda xy.x - y$, ou
- $k = \lambda yx.x - y$.

Motivação

Tais funções, no entanto, podem também ser representadas como funções que retornam outras funções como valores:

$$h^* = \lambda x.(\lambda y.(x - y))$$

De fato, para cada a temos:

$$h^*(a) = (\lambda x.(\lambda y.(x - y))(a) = \lambda y.(a - y)$$

Para cada par a e b temos:

$$(h^*(a))(b) = ((\lambda x.(\lambda y.(x - y))(a))(b) = (\lambda y.(a - y))(b) = a - b = h(a, b)$$

Portanto h^* representa h e, de forma geral, todas as funções com múltiplos parâmetros podem ser representadas através da combinação de funções com um único parâmetro.

Funções computáveis

No Cálculo Lambda, diz-se que uma função $F : \mathbb{N} \to \mathbb{N}$ é computável se e somente se existir uma expressão-lambda f tal que:

$$\forall x, y \in \mathbb{N}, F(x) = y \Leftrightarrow f\,\overline{x} =_\beta \overline{y}$$

onde $\overline{x}$ e $\overline{y}$ são as expressões-lambda que representam, respectivamente, os números naturais x e y. Trata-se apenas de uma das formas possíveis de se definir computabilidade, como é o caso da Máquina de Turing, de outras máquinas, e das funções recursivas. A equivalência desses formalismos foi demonstrada.

Linguagem lambda

Definição

Um λ-termo (também chamado de expressão lambda) é definido de forma indutiva sobre um conjunto de identificadores $\{x, y, z, u, v...\}$ que representam variáveis:

- Uma variável (também chamada "átomo") é um λ-termo;
- Aplicação: se M e N são λ-termos, então (MN) é um λ-termo; representa a aplicação de M a N;
- Abstração: se M é um λ-termo e x é uma variável, então $(\lambda x.M)$ é um λ-termo; representa a função que retorna M com o parâmetro x;

A linguagem lambda é composta de todos os λ-termos que podem ser construídos sobre um certo conjunto de identificadores; trata-se de uma linguagem com apenas dois operadores: aplicação de função a argumentos e abstração.

Linguagem lambda

Gramática

$$
\begin{aligned}
V &\rightarrow u|v|x|y|z|w|... \\
T &\rightarrow V \\
T &\rightarrow (TT) \\
T &\rightarrow (\lambda V.T)
\end{aligned}
$$

Linguagem lambda

Exemplos

São exemplos de λ-termos:

- x
- (xy)
- $(\lambda x.(xy))$
- $((\lambda y.y)(\lambda x.(xy)))$
- $(x(\lambda x.(\lambda x.x)))$
- $(\lambda x.(yz))$

Linguagem lambda

Associatividade e precedência

Para reduzir a quantidade de parênteses, são usadas as seguintes convenções:

- Aplicações tem prioridade sobre abstrações;
- Aplicações são associativas à esquerda;
- Abstrações são associativas à direita.

Por exemplo:

- $\lambda x.PQ$ denota $(\lambda x.(PQ))$ — e não $((\lambda x.P)Q)$;
- $MNPQ$ denota $(((MN)P)Q)$;
- $\lambda xyz.M$ denota $(\lambda x.(\lambda y.(\lambda z.M)))$

O símbolo $\equiv$ será usado para denotar a equivalência sintática de λ-termos.

Linguagem lambda

Exemplos

- $xyz(yx) \equiv (((xy)z)(yx))$
- $\lambda x.(uxy) \equiv (\lambda x.((ux)y))$
- $\lambda u.u(\lambda x.y) \equiv (\lambda u.(u(\lambda x.y)))$
- $(\lambda u.vuu)zy \equiv (((\lambda u.((vu)u))z)y)$
- $ux(yz)(\lambda v.vy) \equiv (((ux)(yz))(\lambda v.(vy)))$
- $(\lambda xyz.xz(yz))uvw \equiv (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))u)v)w)$

Linguagem lambda

Comprimento

O "comprimento" de um λ-termo M — $lgh(M)$ — é o número total de ocorrências de átomos em M.

- Para todo átomo a, $lgh(a) = 1$;
- $lgh(MN) = lgh(M) + lgh(N)$;
- $lgh(\lambda x.M) = 1 + lgh(M)$

Se $M \equiv x(\lambda y.yux)$ então $lgh(M) = 5$.

Linguagem lambda

Ocorrência

Sejam P e Q dois λ-termos. A relação P "ocorre" em Q (ou ainda, P está contido em Q, Q contém P ou P é subtermo de Q) é definida de forma indutiva:

- P ocorre em P;
- Se P ocorre em M ou em N, então P ocorre em (MN);
- Se P ocorre em M ou $P \equiv x$ então P ocorre em $(\lambda x.M)$.

No termo $((xy)(\lambda x.(xy)))$ existem duas ocorrências de (xy) e três de x.

Linguagem lambda

Exemplos

- As ocorrências de xy em $\lambda xy.xy$ são $\lambda xy.xy \equiv (\lambda x.(\lambda y.(\underbrace{xy})))$.
- As ocorrências de uv em $x(uv)(\lambda u.v(\underbrace{uv}))uv$ são $((((x(\underline{uv}))(\lambda u.(v(\underbrace{uv}))))u)v)$.
- O termo $\lambda u.u$ não ocorre em $\lambda u.uv$ pois $\lambda u.uv \equiv (\lambda u.(uv))$.

Linguagem lambda

Escopo

Para uma paticular ocorrência de $\lambda x.M$ em P, a ocorrência de M é chamada de "escopo" da ocorrência de λx à esquerda.
Exemplo: seja

$$P \equiv (\lambda y.yx(\lambda x.y(\lambda y.z)x))vw$$

- O escopo do λy mais à esquerda é $yx(\lambda x.y(\lambda y.z)x)$;
- O escopo do λx é $y(\lambda y.z)x$;
- O escopo do λy mais à direita é z.

Linguagem lambda

Variáveis livres e ligadas

A ocorrência de uma variável x em um termo P é dita:

- "Ligada" se ela está no escopo de um λx em P;
- "Ligada e ligadora" se e somente se ela é o x em λx;
- "Livre" caso contrário.

Linguagem lambda

Variáveis livres e ligadas

- Se x tem pelo menos uma ocorrência ligadora em P, x é chamada de "variável ligada" de P;
- Se x tem pelo menos uma ocorrência livre em P, x é chamada "variável livre" de P;
- O conjunto de todas as variáveis livres de P chamado $FV(P)$;
- Um termo que nao contém variáveis livres é chamado "fechado".

Linguagem lambda

Variáveis livres e ligadas

Para determinar $FV(P)$:

- $FV(\sigma) = \{\sigma\}$ se σ é variável;
- $FV(\sigma) = \emptyset$ se σ é constante;
- $FV((MN)) = FV(M) \cup FV(N)$;
- $FV((\lambda x.M)) = FV(M) - \{x\}$.

Linguagem lambda

Exemplo

Considere o termo $xv(\lambda yz.yv))))w) \equiv$

$$(((xv)(\lambda y.(\lambda z.(yv))))w)$$

- O x mais à esquerda é livre;
- O v mais à esquerda é livre;
- O y mais à esquerda é ligado e ligador;
- O único z é ligado e ligador;
- O y mais à direita é ligado mas não é ligador;
- O v mais à direita é livre;
- O único w é livre.

Linguagem lambda

Exemplo

Considere o termo $P \equiv$

$$(\lambda y.yx(\lambda x.y(\lambda y.z)x))vw$$

- Todos os quatro y são ligados;
- Os y mais à esquerda e mais à direita são ligadores;
- O x mais à esquerda é livre;
- O x central é ligado e ligador;
- O x mais à direita é ligado mas não ligador;
- z, v e w são livres.
- Logo, $FV(P) = \{x, z, v, w\}$; x, nesse caso, é uma variável ligada e também livre de P.

Substituições

Definição

Para todo M, N, x, $[N/x]M$ é definido como o resultado da substituição de toda ocorrência livre de x em M por N, juntamente com a mudança de variáveis ligadas caso isso seja necessário para evitar colisões.

a. $[N/x]x \equiv N$;
b. $[N/x]a \equiv a$, para todo átomo $a \not\equiv x$;
c. $[N/x](PQ) \equiv ([N/x]P[N/x]Q)$;
d. $[N/x](\lambda x.P) \equiv \lambda x.P$;
e. $[N/x](\lambda y.P) \equiv \lambda y.P$, se $x \notin FV(P)$;
f. $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$, se $x \in FV(P)$ e $y \notin FV(N)$;
g. $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$, se $x \in FV(P)$ e $y \in FV(N)$.

Nos casos (e)-(g), $y \not\equiv x$; no caso (g), z é a primeira variável $\notin FV(NP)$.

Substituições

Substituição de variável ligada

Considere (i) $\lambda y.x$ e (ii) $\lambda w.x$. Trata-se da mesma função (função constante que retorna x), porém com diferentes argumentos.

i. Suponha $[w/x](\lambda y.x)$. Então, $[w/x](\lambda y.x) \equiv \lambda y.w$, pela aplicação da regra (f), pois $x \in FV(x)$ e $y \notin FV(w)$;

ii. Suponha $[w/x](\lambda w.x)$. Se a substituição fosse feita também pela regra (f), então $[w/x](\lambda w.x) \equiv \lambda w.w$. Mas $\lambda w.w$ é a função identidade, e não a função constante. Para evitar esse problema, a aplicação da regra (g) produz $[w/x](\lambda w.x) \equiv \lambda z.[w/x][z/w]x \equiv \lambda z.[w/x]x \equiv \lambda z.w$, e nesse caso obtemos a mesma função identidade. Observe que, nesse caso, $x \in FV(x)$ e $w \in FV(w)$.

Substituições

Conversão-α

Seja P um termo que contém uma ocorrência de $\lambda x.M$ e suponha que $y \notin FV(M)$. A substituição de $\lambda x.M$ por

$$\lambda y.[y/x]M$$

é chamada *troca de variável livre* ou ainda *conversão-α* em P. Se P pode ser transformado em Q por meio de uma série finita de conversões-α, diz-se que P e Q são *congruentes* ou então que P é *α-conversível* para Q, denotado

$$P \equiv_\alpha Q.$$

Substituições

Exemplo de conversão-α

$$\begin{aligned} \lambda xy.x(xy) &\equiv \lambda x.(\lambda y.x(xy)) \\ &\equiv_\alpha \lambda x.(\lambda v.x(xv)) \\ &\equiv_\alpha \lambda u.(\lambda v.u(uv)) \\ &\equiv \lambda uv.u(uv)) \end{aligned}$$

Substituições

Propriedades da conversão-α

Para todos P, Q e R:

- *(reflexividade)* $P \equiv_\alpha P$;
- *(transitividade)* $P \equiv_\alpha Q, Q \equiv_\alpha R \Rightarrow P \equiv_\alpha R$;
- *(simetria)* $P \equiv_\alpha Q \Rightarrow Q \equiv_\alpha P$.

Redução-β

Definição

Um termo da forma:

$$(\lambda x.M)N$$

é chamado β-*redex*, e o termo correspondente:

$$[N/x]M$$

é chamado o seu *contractum*. Se um termo P contém uma ocorrência de $(\lambda x.M)N$ e a mesma é substituída por $[N/x]M$, gerando P', diz-se que que ocorrência redex em P foi *contraída* e que P β-*contrai* para P', denotado:

$$P \rhd_{1\beta} P'.$$

Redução-β

Definição

Se um termo P pode ser convertido em um termo Q através de um número finito de reduções-β e conversões-α, diz-se que P β-reduz para Q, denotado:

$$P \rhd_\beta Q.$$

Redução-β

Exemplos

$$(\lambda x.x(xy))N \quad \triangleright_{1\beta} \quad N(Ny)$$
$$(\lambda x.y)N \quad \triangleright_{1\beta} \quad y$$

Redução-β

Exemplos

$$
\begin{aligned}
(\lambda x.(\lambda y.yx)z)v &\rhd_{1\beta} [v/x]((\lambda y.yx)z) \equiv (\lambda y.yv)z \\
&\rhd_{1\beta} [z/y](yv) \equiv zv
\end{aligned}
$$

Redução-β

Exemplos

$$
\begin{aligned}
(\lambda x.xx)(\lambda x.xx) &\rhd_{1\beta} [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
&\rhd_{1\beta} [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
&\rhd_{1\beta} [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
&\rhd_{1\beta} [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
&etc.
\end{aligned}
$$

Redução-β

Exemplos

$$
\begin{aligned}
(\lambda x.xxy)(\lambda x.xxy) &\rhd_{1\beta} (\lambda x.xxy)(\lambda x.xxy)y \\
&\rhd_{1\beta} (\lambda x.xxy)(\lambda x.xxy)yy \\
&\rhd_{1\beta} (\lambda x.xxy)(\lambda x.xxy)yyy \\
&\rhd_{1\beta} (\lambda x.xxy)(\lambda x.xxy)yyyy \\
&etc.
\end{aligned}
$$

Redução-β

Forma normal

- Um termo Q que não possui nenhuma redução-β é chamado de *forma normal-β*;
- Se um termo P reduz-β para um termo Q na forma normal-β, então diz-se que Q é uma *formal normal-β* de P.

Redução-β

Exemplos

- $(\lambda x.(\lambda y.yx)z)v$ tem como forma normal-β zv;
- $L \equiv (\lambda x.xxy)(\lambda x.xxy) \rhd_{1\beta} Ly \rhd_{1\beta} Lyy \rhd_{1\beta}$ ... não tem normal normal-β pois trata-se de uma seqüência infinita e não existe outra forma de reduzir-β a expressão;
- $P \equiv (\lambda u.v)L$ possui as seguintes reduções:
  - $P \equiv (\lambda u.v)L \rhd_{1\beta} [L/u]v \equiv v$;
  - $P \rhd_{1\beta} (\lambda u.v)(Ly) \rhd_{1\beta} (\lambda u.v)(Lyy) \rhd_{1\beta}$ ...

  Portanto, P tem forma normal-β e também uma seqüência infinita de reduções.
- $(\lambda x.xx)(\lambda x.xx)$, também conhecido como Ω, não possui forma normal-β, pois ele reduz sempre para si mesmo e não há outra redução possível.

Redução-β

Interpretação

Expressão lambda:

- Representa um programa, um algoritmo, um procedimento para produzir um resultado;

Redução-β:

- Representa uma computação, a passagem de um estado de um programa para o estado seguinte, dentro do processo de geração de um resultado.

Forma normal:

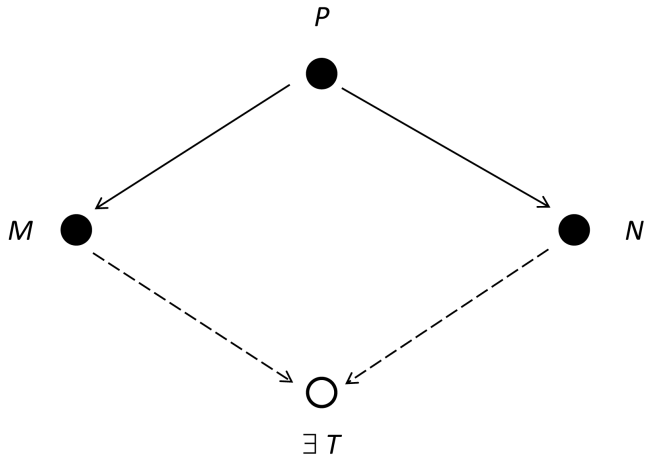
- Representa um resultado de uma computação, um valor que não é passível de novas simplificações ou elaborações.

Redução-β

Teorema de Church-Rosser para $\rhd_\beta$

Se $P \rhd_\beta M$ e $P \rhd_\beta N$, então existe um termo T tal que:

$$M \rhd_\beta T \quad \text{e} \quad N \rhd_\beta T.$$



Redução-β

Teorema de Church-Rosser para $\triangleright_\beta$

A redução-β é *confluente*.
Conseqüências:

- Uma computação no Cálculo Lambda não pode produzir dois ou mais resultados diferentes;
- Duas ou mais reduções de um mesmo termo produzem a mesma forma normal (o resultado da computação independe do caminho escolhido).

Redução-β

Teorema de Church-Rosser para $\rhd_\beta$

Corolário: Se P tem uma forma normal-β, ela é única módulo $\equiv_\alpha$, ou seja, se P possui formas normais-β M e N, então $M \equiv_\alpha N$.

Prova: Suponha que $P \rhd_\beta M$ e $P \rhd_\beta N$, e que ambos M, N reduzem para T. Como M e N não possuem redexes, então $M \equiv_\alpha T$ e $N \equiv_\alpha T$, ou seja, $M \equiv_\alpha N$.

Numerais de Church

Sistemas puro e aplicado

- No Cálculo Lambda *"puro"* não existe representação para números inteiros, operações aritméticas, valores ou operadores lógicos, entre outros tipos de dados e operações usualmente encontradas em linguagens de programação de alto-nível;
- No Cálculo Lambda *"aplicado"* admite-se o uso explícito dos mesmos:

$$\lambda x.x + 1$$

$$(\lambda x.x + 1)(3) \rhd_{1\beta} [3/x](x + 1) \equiv 3 + 1 \equiv 4$$

- É possível, no entanto, representar tipos de dados e operadores quaisquer usando o sistema puro, como demonstram os casos apresentados a seguir.

Numerais de Church

Definição

Para todo $n \in \mathbb{N}$, o "Numeral de Church" de n, denotado $\overline{n}$, é um termo-λ que representa n:

$$\overline{n} := \lambda xy.x^n y$$

onde:

$$x^n y$$

é definido como:

$$\begin{cases} x^n y \equiv \underbrace{x(x(...(x\, y)...))}_{n \text{ vezes}} \text{ se } n \geq 1 \\ x^0 y \equiv y \end{cases}$$

Numerais de Church

Exemplos

$$
\begin{aligned}
\overline{0} &:= \lambda xy.y \\
\overline{1} &:= \lambda xy.xy \\
\overline{2} &:= \lambda xy.x(xy) \\
\overline{3} &:= \lambda xy.x(x(xy)) \\
\overline{4} &:= \lambda xy.x(x(x(xy))) \\
&\ldots
\end{aligned}
$$

Numerais de Church

Propriedade

Os Numerais de Church tem a propriedade de que, para quaisquer termos F e X,

$$\overline{n}FX \rhd_\beta F^n X.$$

Em outras palavras, o numeral de Church inserido na frente de uma aplicação de uma função ao seu argumento representa a aplicação repetida dessa função o mesmo número de vezes.

Numerais de Church

Propriedade

Exemplo:

$$
\begin{array}{lcl}
\overline{2}FX & \equiv & (\lambda xy.x(xy))FX \\
 & \equiv & ((\lambda xy.x(xy))F)X \\
 & \rhd_{1\beta} & [F/x](\lambda y.x(xy))X \\
 & \equiv & (\lambda y.F(Fy))X \\
 & \rhd_{1\beta} & [X/y]F(Fy) \\
 & \equiv & F(FX) \\
 & \equiv & F^2X
\end{array}
$$

Numerais de Church

Sucessor

O sucessor de um Numeral de Church pode ser obtido pela aplicação da expressão:

$$\overline{succ} := \lambda uxy.x(uxy)$$

ao respectivo numeral. É fácil provar que:

$$\overline{succ}\ \overline{n} \rhd_\beta \overline{n+1}.$$

De fato, basta obervar que:

$$(\lambda uxy.x(uxy))\overline{n} \rhd_\beta \lambda xy.x(\overline{n}xy) \equiv \lambda x.\lambda y.xx^n y \equiv \lambda x.\lambda y.x^{n+1}y \equiv \overline{n+1}.$$

Numerais de Church

Sucessor

Exemplo:

$$
\begin{array}{rcl}
\overline{succ}\ \overline{0} & \equiv & (\lambda uxy.x(uxy))(\lambda xy.y) \\
& \rhd_{1\beta} & [(\lambda xy.y)/u](\lambda xy.x(uxy)) \\
& \equiv & (\lambda xy.x((\lambda xy.y)xy) \\
& \rhd_{\beta} & (\lambda xy.xy) \\
& \equiv & \overline{1}
\end{array}
$$

Numerais de Church

Adição

A adição de dois Numerais de Church pode ser obtida pela aplicação da expressão:

$$\overline{add} := \lambda uvxy.ux(vxy)$$

aos respectivos operandos. Nesse caso, temos que:

$$\overline{add}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m+n}.$$

De fato, basta observar que:

$$(\lambda uvxy.ux(vxy))\overline{m}\ \overline{n} \rhd_\beta \lambda xy.\overline{m}x(\overline{n}xy) \equiv \lambda xy.\overline{m}x(x^n y)$$

$$\equiv \lambda xy.x^m(x^n y) \equiv \lambda xy.x^{m+n}y \equiv \overline{m+n}$$

Numerais de Church

Adição

Exemplo:

$$
\begin{aligned}
\overline{add}\ \overline{1}\ \overline{2} &\equiv (\lambda uvxy.ux(vxy))(\lambda xy.xy)(\lambda xy.x(xy)) \\
&\rhd_{1\beta} ([(\lambda xy.xy)/u](\lambda vxy.ux(vxy)))(\lambda xy.x(xy)) \\
&\equiv (\lambda vxy.(\lambda xy.xy)x(vxy))(\lambda xy.x(xy)) \\
&\rhd_{\beta} (\lambda vxy.x(vxy))(\lambda xy.x(xy)) \\
&\rhd_{1\beta} [(\lambda xy.x(xy))/v](\lambda xy.x(vxy)) \\
&\equiv (\lambda xy.x((\lambda xy.x(xy))xy)) \\
&\rhd_{\beta} (\lambda xy.x(x(xy))) \\
&\equiv \overline{3}
\end{aligned}
$$

Numerais de Church

Multiplicação

A multiplicação de dois Numerais de Church pode ser obtida pela aplicação da expressão:

$$\overline{mult} := \lambda uvx.u(vx)$$

aos respectivos operandos. Nesse caso, temos que:

$$\overline{mult}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m * n}$$

.

Numerais de Church

Multiplicação

De fato, temos que:

$$
\begin{array}{lcl}
(\lambda uvx.u(vx))\ \overline{m}\ \overline{n} & \rhd_\beta & \lambda x.\overline{m}(\overline{n}x) \\
& \equiv & \lambda x.\overline{m}((\lambda y.\lambda z.y^n z)x) \\
& \rhd_\beta & \lambda x.\overline{m}(\lambda z.x^n z) \\
& \equiv & \lambda x.(\lambda u.\lambda v.u^m v)(\lambda z.x^n z) \\
& \rhd_\beta & \lambda x.[\lambda z.x^n z/u](\lambda v.u^m v) \\
& \equiv & \lambda x.\lambda v.(\lambda z.x^n z)^m v \\
& \equiv & \lambda x.\lambda v.(\lambda z.x^n z)^{m-1}((\lambda z.x^n z)v)
\end{array}
$$

Numerais de Church

Multiplicação

Continuação:

$$
\begin{aligned}
\lambda x.\lambda v.(\lambda z.x^n z)^{m-1}((\lambda z.x^n z)v) &\rhd_\beta \lambda x.\lambda v.(\lambda z.x^n z)^{m-1}(x^n v) \\
&\equiv \lambda x.\lambda v.(\lambda z.x^n z)^{m-2}((\lambda z.x^n z)x^n v) \\
&\rhd_\beta \lambda x.\lambda v.(\lambda z.x^n z)^{m-2}(x^n(x^n v)) \\
&\equiv \lambda x.\lambda v.(\lambda z.x^n z)^{m-2}(x^{2*n} v)) \\
&\rhd_\beta \lambda x.\lambda v.x^{m*n} v \\
&\equiv \overline{m*n}
\end{aligned}
$$

Numerais de Church

Multiplicação

Exemplo:

$$
\begin{aligned}
\overline{mult}\ \overline{2}\ \overline{2} &\equiv (\lambda uvx.u(vx))\overline{2}\ \overline{2} \\
&\rhd_{1\beta} ([\overline{2}/u](\lambda vx.u(vx)))\overline{2} \\
&\equiv (\lambda vx.\overline{2}(vx)))\overline{2} \\
&\rhd_{1\beta} [\overline{2}/v](\lambda x.\overline{2}(vx)) \\
&\equiv \lambda x.\overline{2}(\overline{2}x) \\
&\rhd_{1\beta} \lambda x.\overline{2}(\lambda y.x(xy)) \\
&\rhd_{1\beta} \lambda x.\lambda y.(\lambda y.x(xy))((\lambda y.x(xy))y \\
&\rhd_{1\beta} \lambda x.\lambda y.(\lambda y.x(xy))(x(xy) \\
&\rhd_{1\beta} \lambda x.\lambda y.x(x(x(xy))) \\
&\equiv \overline{4}
\end{aligned}
$$

Numerais de Church

Exponenciação

A exponenciação de dois Numerais de Church pode ser obtida pela aplicação da expressão:

$$\overline{exp} := \lambda uv.vu$$

aos respectivos operandos. Nesse caso, temos que:

$$\overline{exp}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m^n}.$$

Numerais de Church

Exponenciação

De fato, temos que:

$$
\begin{aligned}
(\lambda uv.vu)\ \overline{m}\ \overline{n} &\rhd_\beta \overline{n}\ \overline{m} \\
&\equiv (\lambda x.\lambda y.x^n y)\ \overline{m} \\
&\rhd_\beta \lambda y.(\overline{m})^n y \\
&\equiv \lambda y.(\overline{m}(\overline{m}(...(\overline{m}(\overline{m}y))))) \\
&\rhd_\beta \lambda y.(\overline{m}(\overline{m}(...(\overline{m}(\lambda w.y^m w))))) \\
&\rhd_\beta \lambda y.(\overline{m}(\overline{m}(...(\lambda w.y^{m^2} w)))) \\
&\rhd_\beta \lambda y.(\lambda w.y^{m^n} w) \\
&\equiv \overline{m^n}
\end{aligned}
$$

Numerais de Church

Exponenciação

Exemplo:

$$
\begin{array}{rcl}
\overline{exp}\ \overline{2}\ \overline{2} & \equiv & (\lambda u.\lambda v.vu)\overline{2}\ \overline{2} \\
& \rhd_\beta & (\lambda v.v\overline{2})\overline{2} \\
& \rhd_\beta & \overline{2}\ \overline{2} \\
& \equiv & (\lambda x.\lambda y.x(xy))\overline{2} \\
& \rhd_\beta & \lambda y.\overline{2}(\overline{2}y) \\
& \equiv & \lambda y.\overline{2}((\lambda x.\lambda z.x.(xz))y) \\
& \rhd_\beta & \lambda y.\overline{2}(\lambda z.y(yz)) \\
& \equiv & \lambda y.(\lambda x.\lambda w.x(xw))(\lambda z.y(yz))
\end{array}
$$

Numerais de Church

Exponenciação

Exemplo (continuação):

$$
\begin{aligned}
\lambda y.(\lambda x.\lambda w.x(xw))(\lambda z.y(yz)) &\rhd_\beta \lambda y.[\lambda z.y(yz)/x](\lambda w.x(xw)) \\
&\equiv \lambda y.\lambda w.[\lambda z.y(yz)/x](x(xw)) \\
&\equiv \lambda y.\lambda w.(\lambda z.y(yz))((\lambda z.y(yz))w) \\
&\rhd_\beta \lambda y.\lambda w.(\lambda z.y(yz))(y(yw)) \\
&\rhd_\beta \lambda y.\lambda w.[y(yw)/z](y(yz)) \\
&\equiv \lambda y.\lambda w.(y(y(y(yw)))) \\
&\equiv \overline{4}
\end{aligned}
$$

Numerais de Church

Outras operações

- Predecessor ($n-1$ se $n>0$ ou 0 caso contrário):

$$\overline{pred} := \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u)$$

- Subtração ($m-n$ se $m \geq n$ ou 0 caso contrário):

$$\overline{sub} := \lambda m.\lambda n.(n\ \overline{pred})m$$

Numerais de Church

Expressões compostas

Através da combinação das expressões lambda anteriores, é possível representar expressões aritméticas mais complexas, como é o caso de:

$$(2^3 + 4) * 5$$

que é denotada:

$$\overline{mult}\ (\ \overline{add}\ (\ \overline{exp}\ \overline{2}\ \overline{3}\ )\ \overline{4}\ )\ \overline{5} \equiv$$

$$\underbrace{\underbrace{(\lambda uvx.u(vx))}_{\overline{mult}}\ \underbrace{\underbrace{(\lambda uvxy.ux(vxy))}_{\overline{add}}\ \underbrace{\underbrace{(\lambda uv.vu)}_{\overline{exp}}\ \overline{2}\ \overline{3}}_{2^3})\ \overline{4}}_{2^3+4})\ \overline{5} \rhd_\beta \overline{60}.}_{(2^3+4)*5}$$

Booleanos de Church

Definição

O Cálculo Lambda puro também permite a representação de valores e operações lógicas:

- $\overline{true} := \lambda x.\lambda y.x$
  (projeção do primeiro argumento);
- $\overline{false} := \lambda x.\lambda y.y$
  (projeção do segundo argumento);
  Observar que $\overline{false} \equiv \overline{0}$.

Booleanos de Church

AND

$$\overline{and} := \lambda x.\lambda y.xyx$$

É possível provar que:

$$\overline{and}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m\ and\ n}$$

De fato:

$$(\lambda x.\lambda y.xyx)\ \overline{m}\ \overline{n} \rhd_\beta \overline{m}\ \overline{n}\ \overline{m}$$

- Se $m = TRUE$, então projeta como resultado o valor de n;
- Se $m = FALSE$, projeta como resultado o próprio valor de m.

Booleanos de Church

AND

Exemplos:

$$(\lambda x.\lambda y.xyx)\ \overline{true}\ \overline{true} \quad \rhd_\beta \quad \overline{true}\ \underbrace{\overline{true}}\ \overline{true}$$

$$\rhd_\beta \quad \overline{true}$$

$$(\lambda x.\lambda y.xyx)\ \overline{false}\ \overline{true} \quad \rhd_\beta \quad \overline{false}\ \overline{true}\ \underbrace{\overline{false}}$$

$$\rhd_\beta \quad \overline{false}$$

Booleanos de Church

OR

$$\overline{or} := \lambda x.\lambda y.xxy$$

É possível provar que:

$$\overline{or}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m\ or\ n}$$

De fato:

$$(\lambda x.\lambda y.xxy)\ \overline{m}\ \overline{n} \rhd_\beta \overline{m}\ \overline{m}\ \overline{n}$$

- Se $m = TRUE$, então projeta como resultado o próprio valor de m;
- Se $m = FALSE$, projeta como resultado o valor de n.

Booleanos de Church

OR

Exemplos:

$$(\lambda x.\lambda y.xxy)\ \overline{true}\ \overline{false} \rhd_\beta \overline{true}\ \underbrace{\overline{true}}\ \overline{false}$$
$$\rhd_\beta \overline{true}$$

$$(\lambda x.\lambda y.xxy)\ \overline{false}\ \overline{true} \rhd_\beta \overline{false}\ \overline{false}\ \underbrace{\overline{true}}$$
$$\rhd_\beta \overline{true}$$

Booleanos de Church

NOT

$$\overline{not} := \lambda x.\lambda y.\lambda z.xzy$$

É possível provar que:

$$\overline{not}\ \overline{m} \rhd_\beta \overline{not(m)}$$

De fato:

$$(\lambda x.\lambda y.\lambda z.xzy)\ \overline{m} \rhd_\beta \lambda y.\lambda z.\overline{m}zy$$

- Se $m = TRUE$, então $\lambda y.\lambda z.\overline{m}zy \rhd_\beta \lambda y.\lambda z.z \equiv \overline{false}$;
- Se $m = FALSE$, então $\lambda y.\lambda z.\overline{m}zy \rhd_\beta \lambda y.\lambda z.y \equiv \overline{true}$.

Booleanos de Church

NOT

Exemplos:

$$\begin{aligned}(\lambda x.\lambda y.\lambda z.xzy)\ \overline{true} &\rhd_\beta \lambda y.\lambda z.(\overline{true})zy \\ &\rhd_\beta \lambda y.\lambda z.z \\ &\equiv \overline{false}\end{aligned}$$

$$\begin{aligned}(\lambda x.\lambda y.\lambda z.xzy)\ \overline{false} &\rhd_\beta \lambda y.\lambda z.(\overline{false})zy \\ &\rhd_\beta \lambda y.\lambda z.y \\ &\equiv \overline{true}\end{aligned}$$

Booleanos de Church

XOR

$$\overline{xor} := \lambda x.\lambda y.\lambda z.\lambda w.x(ywz)(yzw)$$

É possível provar que:

$$\overline{xor}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m\ xor\ n}$$

Booleanos de Church

XOR

De fato:

$$(\lambda x.\lambda y.\lambda z.\lambda w.x(ywz)(yzw))\ \overline{m}\ \overline{n} \rhd_\beta \lambda z.\lambda w.\overline{m}(\overline{n}wz)(\overline{n}zw)$$

- Se $m = TRUE$, então $\lambda z.\lambda w.\overline{m}(\overline{n}wz)(\overline{n}zw) \rhd_\beta \lambda z.\lambda w.\overline{n}wz$;
  - Se $n = TRUE$, então $\lambda z.\lambda w.\overline{n}wz \rhd_\beta \lambda z.\lambda w.w \equiv \overline{false}$;
  - Se $n = FALSE$, então $\lambda z.\lambda w.\overline{n}wz \rhd_\beta \lambda z.\lambda w.z \equiv \overline{true}$.
- Se $m = FALSE$, então $\lambda z.\lambda w.\overline{m}(\overline{n}wz)(\overline{n}zw) \rhd_\beta \lambda z.\lambda w.\overline{n}zw$;
  - Se $n = TRUE$, então $\lambda z.\lambda w.\overline{n}zw \rhd_\beta \lambda z.\lambda w.z \equiv \overline{true}$;
  - Se $n = FALSE$, então $\lambda z.\lambda w.\overline{n}zw \rhd_\beta \lambda z.\lambda w.w \equiv \overline{false}$.

Booleanos de Church

XOR

Exemplos:

$$
\begin{aligned}
(\lambda x.\lambda y.\lambda z.\lambda w.x(ywz)(yzw))\ \overline{true}\ \overline{false} &\rhd_\beta \\
\lambda z.\lambda w.\overline{true}((\overline{false})wz)((\overline{false}))zw &\rhd_\beta \\
\lambda z.\lambda w.(\overline{false})wz &\rhd_\beta \\
\lambda z.\lambda w.z &\equiv \overline{true}
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.\lambda y.\lambda z.\lambda w.x(ywz)(yzw))\ \overline{false}\ \overline{false} &\rhd_\beta \\
\lambda z.\lambda w.\overline{false}((\overline{false})wz)((\overline{false}))zw &\rhd_\beta \\
\lambda z.\lambda w.(\overline{false})zw &\rhd_\beta \\
\lambda z.\lambda w.w &\equiv \overline{false}
\end{aligned}
$$

Booleanos de Church

IF

$$\overline{if} := \lambda x.\lambda y.\lambda z.xyz$$

É possível provar que:

$$\overline{if}\ \overline{e}\ \overline{m}\ \overline{n} \rhd_\beta \overline{m} \text{ se } e = TRUE$$

$$\overline{if}\ \overline{e}\ \overline{m}\ \overline{n} \rhd_\beta \overline{n} \text{ se } e = FALSE$$

De fato:

$$(\lambda x.\lambda y.\lambda z.xyz)\ \overline{e}\ \overline{m}\ \overline{n} \rhd_\beta \overline{e}\ \overline{m}\ \overline{n}$$

- Se $e = TRUE$, então projeta m como resultado;
- Se $e = FALSE$, projeta n como resultado.

Booleanos de Church

IF

Exemplos:

$$(\lambda x.\lambda y.\lambda z.xyz)\ \overline{true}\ \overline{m}\ \overline{n} \quad \triangleright_\beta \quad \overline{true}\ \overline{m}\ \overline{n}$$
$$\triangleright_\beta \quad \overline{m}$$

$$(\lambda x.\lambda y.\lambda z.xyz)\ \overline{false}\ \overline{m}\ \overline{n} \quad \triangleright_\beta \quad \overline{false}\ \overline{m}\ \overline{n}$$
$$\triangleright_\beta \quad \overline{n}$$

Booleanos de Church

Expressões compostas - ZERO

Através da combinação das expressões anteriores, é possível representar funções mais complexas, que fazem uso de valores e operadores lógicos e aritméticos simultâneamente, como é o caso da função que testa se o argumento é zero e retorna $\overline{true}$ ou $\overline{false}$:

$$\overline{zero} := \lambda x.x(\lambda y.\overline{false})\ \overline{true}.$$

Booleanos de Church

Expressões compostas - ZERO

De fato, para $n = 0$:

$$\begin{aligned} (\lambda x.x(\lambda y.\overline{false})\ \overline{true})\ \overline{0} &\ \rhd_\beta \\ \overline{0}\ (\lambda y.\overline{false})\ \overline{true} &\ \equiv \\ \overline{false}\ (\lambda y.\overline{false})\ \overline{true} &\ \rhd_\beta \\ \overline{true} & \end{aligned}$$

Booleanos de Church

Expressões compostas - ZERO

E para $n > 0$:

$$
\begin{aligned}
(\lambda x.x(\lambda y.\overline{false})\ \overline{true})\ \overline{n} &\ \rhd_\beta \\
\overline{n}\ (\lambda y.\overline{false})\ \overline{true} &\ \equiv \\
(\lambda z.\lambda w.z^n w)(\lambda y.\overline{false})\ \overline{true} &\ \rhd_\beta \\
(\lambda w.(\lambda y.\overline{false})^n w)\ \overline{true} &\ \rhd_\beta \\
(\lambda w.(\lambda y.\overline{false})^{n-1}((\lambda y.\overline{false})w))\ \overline{true} &\ \rhd_\beta \\
(\lambda w.(\lambda y.\overline{false})^{n-1}\ \overline{false})\ \overline{true} &\ \rhd_\beta \\
\ldots & \\
(\lambda w.(\lambda y.\overline{false})\ \overline{false})\ \overline{true} &\ \rhd_\beta \\
(\lambda w.\overline{false})\ \overline{true} &\ \rhd_\beta \\
\overline{false} &
\end{aligned}
$$

Booleanos de Church

Expressões compostas - LEQ

Função que testa se o primeiro argumento é menor ou igual que o segundo:

$$\overline{leq} := \lambda x.\lambda y.\overline{zero}\ (\overline{sub}\ \overline{x}\ \overline{y}).$$

De fato:

$$\begin{aligned}(\lambda x.\lambda y.\overline{zero}\ (\overline{sub}\ \overline{x}\ \overline{y}))\ \overline{m}\ \overline{n} &\quad \rhd_\beta \\ \overline{zero}\ (\overline{sub}\ \overline{m}\ \overline{n}) &\quad \rhd_\beta \\ \overline{zero}\ (\overline{m-n}) &\end{aligned}$$

Agregados

Pares ordenados

O Cálculo Lambda puro permite a representação de pares ordenados:

$$\overline{pair} := \lambda x.\lambda y.\lambda z.zxy$$

A seleção dos elementos é feita através das funções:

- $\overline{first} := \lambda p.(p\,\overline{true})$
  (seleção do primeiro elemento);
- $\overline{second} := \lambda p.(p\,\overline{false})$
  (seleção do segundo elemento);

Agregados

Pares ordenados

Exemplos:

$$\begin{aligned}
\overline{pair}\,\overline{1}\,\overline{2} &\rhd_\beta (\lambda xyz.zxy)\,\overline{1}\,\overline{2} \\
&\rhd_\beta \lambda z.\overline{1}\,\overline{2}
\end{aligned}$$

$$\begin{aligned}
\overline{first}\,(\overline{pair}\,\overline{1}\,\overline{2}) &\rhd_\beta (\lambda p.p\,\overline{true})(\lambda z.\overline{1}\,\overline{2}) \\
&\rhd_\beta (\lambda z.z\overline{1}\,\overline{2})\,\overline{true} \\
&\rhd_\beta \overline{true}\,\overline{1}\,\overline{2} \\
&\rhd_\beta \overline{1}
\end{aligned}$$

$$\begin{aligned}
\overline{second}\,(\overline{pair}\,\overline{1}\,\overline{2}) &\rhd_\beta (\lambda p.p\,\overline{false})(\lambda z.\overline{1}\,\overline{2}) \\
&\rhd_\beta (\lambda z.z\overline{1}\,\overline{2})\,\overline{false} \\
&\rhd_\beta \overline{false}\,\overline{1}\,\overline{2} \\
&\rhd_\beta \overline{2}
\end{aligned}$$

Igualdade-β

Definição

Um termo P é dito "β-igual" ou "β-conversível" a um termo Q, denotado $P =_\beta Q$ se e somente se Q puder ser obtido a partir de P por uma seqüência finita (eventualmente vazia) de contrações-β, contrações-β reversas e mudanças de variáveis ligadas.
Ou seja, $P =_\beta Q$ se e somente se existir $P_0, ..., P_n (n \geq 0)$ tal que:

$$(\forall i \leq n-1)(P_i \rhd_{1\beta} P_{i+1} \quad \text{ou} \quad P_{i+1} \rhd_{1\beta} P_i \quad \text{ou} \quad P_i =_\alpha P_{i+1},$$

$$P_0 \equiv P,$$

$$P_n \equiv Q.$$

Igualdade-β

Exemplo

Sejam $P \equiv (\lambda xyz.xzy)(\lambda xy.x)$ e $Q \equiv (\lambda xy.x)(\lambda x.x)$.
Então $P =_\beta Q$, ou seja:

$$(\lambda xyz.xzy)(\lambda xy.x) =_\beta (\lambda xy.x)(\lambda x.x)$$

De fato, pode-se notar inicialmente que:

$$\begin{aligned}
(\lambda xyz.xzy)(\lambda xy.x) &\equiv_\alpha (\lambda xyz.xzy)(\lambda uv.u) \\
&\rhd_{1\beta} \lambda yz.(\lambda uv.u)zy \\
&\rhd_{1\beta} \lambda yz.(\lambda v.z)y \\
&\rhd_{1\beta} \lambda yz.z
\end{aligned}$$

Igualdade-β

Exemplo

Além disso, que:

$$\begin{aligned}(\lambda xy.x)(\lambda x.x) &\equiv_\alpha (\lambda xy.x)(\lambda w.w)\\ &\rhd_{1\beta} \lambda y.(\lambda w.w)\\ &\equiv \lambda yw.w\\ &\equiv_\alpha \lambda yz.z\end{aligned}$$

Igualdade-β

Exemplo

Finalmente, pode-se considerar $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$ tais que:

$$\begin{aligned} P = P_0 &= (\lambda xyz.xzy)(\lambda xy.x) \\ P_1 &= (\lambda xyz.xzy)(\lambda uv.u) \\ P_2 &= \lambda yz.(\lambda uv.u)zy \\ P_3 &= \lambda yz.(\lambda v.z)y \\ P_4 &= \lambda yz.z \\ P_5 &= \lambda yw.w \\ P_6 &= (\lambda xy.x)(\lambda w.w) \\ Q = P_7 &= (\lambda xy.x)(\lambda x.x) \end{aligned}$$

Para concluir que $P =_\beta Q$, basta observar que $P_0 \equiv_\alpha P_1$, $P_1 \rhd_{1\beta} P_2$, $P_2 \rhd_{1\beta} P_3$, $P_3 \rhd_{1\beta} P_4$, $P_4 \equiv_\alpha P_5$, $P_6 \rhd_{1\beta} P_5$ e $P_6 \equiv_\alpha P_7$.

Igualdade-β

Lemas

Lema: $P =_\beta Q, P \equiv_\alpha P', Q \equiv_\alpha Q' \Rightarrow P' =_\beta Q'$.

Lema: $M =_\beta M', N =_\beta N' \Rightarrow [N/x]M =_\beta [N'/x]M'$.

Igualdade-β

Teorema de Church-Rosser para $=_\beta$

Se $P =_\beta Q$, então existe um termo T tal que:

$$P \rhd_\beta T \quad \text{e} \quad Q \rhd_\beta T.$$

Dois termos β-conversíveis representam a mesma operação, uma vez que ambos podem ser reduzidos para o mesmo termo.

Igualdade-β

Corolários

Corolário: Se $P =_\beta Q$ e Q é uma forma normal-β, então $P \rhd_\beta Q$

Corolário: Se $P =_\beta Q$, então ambos P e Q possuem a mesma forma normal-β ou então nenhum dos dois possui nenhuma forma normal-β.

Corolário: Se P e Q são formas normais-β e $P =_\beta Q$, então $P \equiv_\alpha Q$.

Corolário (*unicidade da forma normal*): Um termo é β-igual a no máximo uma forma normal-β, a menos de mudanças de variáveis ligadas.

Problems

Some computations may not terminate:

$$
\begin{array}{rcl}
(\lambda x.xx)(\lambda x.xx) & \rhd_\beta & [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
 & \rhd_\beta & [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
 & \rhd_\beta & [(\lambda x.xx)/x](xx) \equiv (\lambda x.xx)(\lambda x.xx) \\
 & \ldots & etc. \\
(\lambda x.xxy)(\lambda x.xxy) & \rhd_\beta & (\lambda x.xxy)(\lambda x.xxy)y \\
(\lambda x.xxy)(\lambda x.xxy) & \rhd_\beta & (\lambda x.xxy)(\lambda x.xxy)yy \\
(\lambda x.xxy)(\lambda x.xxy) & \rhd_\beta & (\lambda x.xxy)(\lambda x.xxy)yyy \\
 & \ldots & etc.
\end{array}
$$

Characteristics

- Created by Church to avoid the inconsistencies of the untyped version;
- Type tags are associated to lambda terms;
- Variables have base types $(x : \sigma)$;
- Abstractions and applications create new types accordingly;
- Types must match;
- Less powerful model of computation;
- Type systems for programming languages;
- Equality of terms is decidable;
- Strongly normalizing (all computations terminate);
- $(\lambda x.xx)(\lambda x.xx)$ and $(\lambda x.xxy)(\lambda x.xxy)$ are not terms of the typed lambda calculus.

Inference rules for abstraction and application

Abstraction ("function type"):

$$\frac{\begin{array}{c}[x:\sigma]\\ M:\tau\end{array}}{\lambda x^{\sigma}.M:\sigma\to\tau}\ (\to I)$$

$$(\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau}$$

Application:

$$\frac{M:\sigma\to\tau \qquad N:\sigma}{MN:\tau}\ (\to E)$$

$$(M^{\sigma\to\tau}N^{\sigma})^{\tau}$$

Example 1: type tree

Type:

$$(a \to (b \to c)) \to (b \to (a \to c))$$

Term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to (b \to c) \qquad z : a}{xz : b \to c}(\to E) \qquad y : b}{xzy : c}(\to E)}{\lambda z^{a}.xzy : (a \to c)}(\to I)}{\lambda y^{b}.\lambda z^{a}.xzy : (b \to (a \to c))}(\to I)}{\lambda x^{a \to (b \to c)}.\lambda y^{b}.\lambda z^{a}.xzy : (a \to (b \to c)) \to (b \to (a \to c))}(\to I)$$

Inference rules for conjunction ($\times$)

Introduction:

$$\frac{x : \sigma \qquad y : \tau}{\overline{conj}\, xy : \sigma \times \tau}\ (\times I)$$

$$(\overline{conj}\, xy \equiv \overline{pair}\, xy)$$

Elimination 1:

$$\frac{p : \sigma \times \tau}{\overline{first}\, p : \sigma}\ (\times E_1)$$

Elimination 2:

$$\frac{p : \sigma \times \tau}{\overline{second}\, p : \tau}\ (\times E_2)$$

Example 2: type tree

Type:

$$(a \times b) \to (b \times a)$$

Term:

$$\dfrac{\dfrac{\dfrac{x : a \times b}{\overline{second}\, x : b}\,(\times E) \qquad \dfrac{x : a \times b}{\overline{first}\, x : a}\,(\times E)}{\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : b \times a}\,(\times I)}{\lambda x^{a \times b}.\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : (a \times b) \to (b \times a)}\,(\to I)$$

Inference rules for disjunction (+)

Introduction 1:

$$\frac{x : \sigma}{\overline{inl}\, x : \sigma + \tau}\,(+I_1)$$

$$(\overline{inl}\, x \equiv \overline{pair}\, \overline{0}\, x)$$

Introduction 2:

$$\frac{y : \tau}{\overline{inr}\, y : \sigma + \tau}\,(+I_2)$$

$$(\overline{inr}\, y \equiv \overline{pair}\, \overline{1}\, y)$$

Inference rules for disjunction (+)

Elimination:

$$\frac{\begin{array}{ccc} & [x:\sigma] & [y:\tau] \\ & \dots & \dots \\ p:\sigma+\tau & q:\mu & r:\mu \end{array}}{\overline{case}\,p\,(\lambda x.q)\,(\lambda y.r):\mu}\,(+E)$$

$$(\overline{case}\,u\,v\,w \equiv \overline{if}\,(\overline{zero}\,(\overline{first}\,u))(v\,(\overline{second}\,u))(w\,(\overline{second}\,u)))$$

Example 3: type tree

Type:

$$(a + (a \times b)) \to a$$

Term:

$$\frac{\dfrac{p : a + (a \times b) \qquad \begin{array}{c}[x : a] \\ x : a\end{array} \qquad \dfrac{[y : a \times b]}{\overline{first}\, y : a}\,(\times E)}{\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y) : a}\,(+E)}{\lambda p^{a+(a\times b)}.(\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y)) : (a + (a \times b)) \to a}\,(\to I)$$

Inference rules for false ($\bot$)

Introduction:

No rule.

Elimination (*ex falso quodlibet*):

$$\frac{x : \bot}{\lambda \bot . x^{\bot} : P}\ (\bot E)$$

Inference rules for negation ($\neg$)

Introduction (same as implication introduction):

$$\frac{\begin{array}{c} x : P \\ \dots \\ f : \bot \end{array}}{\lambda x^P . f : \neg P} \ (\neg I, \text{ same as } \Rightarrow I)$$

Elimination (same as implication elimination):

$$\frac{x : A \qquad y : \neg A}{yx : \bot} \ (\neg E, \text{ same as } \Rightarrow E)$$

Example 4: type tree

Type:

$$(a \to b) \to (\neg b \to \neg a)$$

Term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to b \qquad y : a}{xy : b}(\to E) \qquad z : \neg b}{z(xy) : \bot}(\neg E)}{\lambda y^{a}.z(xy) : \neg a}(\to I)}{\lambda z^{\neg b}.\lambda y^{a}.z(xy) : \neg b \to \neg a}(\to I)}{\lambda x^{a \to b}.\lambda z^{\neg b}.\lambda y^{a}.z(xy) : (a \to b) \to (\neg b \to \neg a)}(\to I)$$

Inference rules for universal quantifier ($\forall$)

Introduction:

$$\frac{\begin{array}{c}[x : D] \\ \dots \\ f : P(x)\end{array}}{\lambda x^D.f : \forall x.P(x)}\ (\forall I)$$

Elimination:

$$\frac{t : D \qquad r : \forall x.P(x)}{rt : P(t)}\ (\forall E)$$

Inference rules for existential quantifier ($\exists$)

Introduction:

$$\frac{a : D \qquad f(a) : P(a)}{\varepsilon x.(f(x), a)) : \exists x.P(x)}\ (\exists I)$$

Elimination:

$$\frac{r : \exists x.P(x) \qquad \begin{array}{c}[t : D, g(t) : P(t)] \\ \cdots \\ h(t, g) : C\end{array}}{E(r, \lambda g.\lambda t.h(t, g)) : C}\ (\exists E)$$

Example 5: type tree

Type:

$$\forall x.R(x,x) \to \forall x.\exists y.R(x,y)$$

Term:

$$\dfrac{\dfrac{\dfrac{\dfrac{t : D \qquad r : \forall x.R(x,x)}{rt : R(t,t)}\,(\forall E) \qquad t : D}{\varepsilon y.(ry,t) : \exists y.R(t,y)}\,(\exists I)}{\lambda t.\varepsilon y.(ry,t) : \forall t.\exists y.R(t,y)}\,(\forall I)}{\lambda r.\lambda t.\varepsilon y.(ry,t) : \forall x.R(x,x) \to \forall t.\exists y.R(t,y)}\,(\to I)$$

Reasoning × Computing

Mathematics is all about:

- Reasoning;
- Computing.

For long time considered as separate areas; even today, ignored by many.
Any relation there?

Reasoning × Computing

YES, according to the Curry-Howard Isomorphism.

- There is a direct relationship between reasoning (as expressed by first-order logic and natural deduction) and computing (as expressed by the typed lambda calculus);
- *Proofs-as-programs* or *Propositions-as-types* notions;
- First observed by (Haskell) Curry in 1934, later developed and extended by Curry in 1958 and William Howard in 1969;

Reasoning × Computing

- This has many important consequences as is the basis of modern software development and computer assisted theorem proofing:
  - Reasoning principles and techniques can be brought into software development;
  - Computing (idem) can be used in theorem proving.
- In the *simply typed lambda calculus*, the function operator ($\rightarrow$) corresponds to the implication connective ($\Rightarrow$); correspondences also exist for other operators.

The Isomorphism

General picture:

Proofs	Theorems
Programs	Types

Proofs & Theorems

First of all:

Proofs $\Leftrightarrow$ Theorems

Programs Types

Proofs & Theorems

Example 1

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow (b \Rightarrow c) \qquad a}{b \Rightarrow c}\,(\Rightarrow E) \qquad b}{c}\,(\Rightarrow E)}{a \Rightarrow c}\,(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

Theorem:

$$(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$$

Proofs & Theorems

Example 2

Proof:

$$\dfrac{\dfrac{\dfrac{a \wedge b}{b}\,(\wedge E) \qquad \dfrac{a \wedge b}{a}\,(\wedge E)}{b \wedge a}\,(\wedge I)}{(a \wedge b) \Rightarrow (b \wedge a)}\,(\Rightarrow I)$$

Theorem:

$$(a \wedge b) \Rightarrow (b \wedge a)$$

Proofs & Theorems

Example 3

Proof:

$$\dfrac{\dfrac{a \vee (a \wedge b) \qquad \begin{matrix}[a] \\ a\end{matrix} \qquad \dfrac{[a \wedge b]}{a}\,(\wedge E)}{a}\,(\vee E)}{(a \vee (a \wedge b)) \Rightarrow a}\,(\Rightarrow I)$$

Theorem:

$$(a \vee (a \wedge b)) \Rightarrow a$$

Proofs & Theorems

Example 4

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow b \qquad a}{b}\,(\Rightarrow E) \qquad \neg b}{\bot}\,(\neg E)}{\neg a}\,(\Rightarrow I)}{\neg b \Rightarrow \neg a}\,(\Rightarrow I)}{(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)}\,(\Rightarrow I)$$

Theorem:

$$(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)$$

Proofs & Theorems

Example 5

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{t : D \qquad r : \forall x.R(x,x)}{rt : R(t,t)}(\forall E) \qquad t : D}{\varepsilon y.(ry,t) : \exists y.R(t,y)}(\exists I)}{\lambda t.\varepsilon y.(ry,t) : \forall t.\exists y.R(t,y)}(\forall I)}{\lambda r.\lambda t.\varepsilon y.(ry,t) : \forall x.R(x,x) \Rightarrow \forall t.\exists y.R(t,y)}(\Rightarrow I)$$

Theorem:

$$\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$$

Programs & Types

Also:

Proofs		Theorems
Programs	$\Leftrightarrow$	Types

Programs & Types

Example 1

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to (b \to c) \qquad z : a}{xz : b \to c}(\to E) \qquad y : b}{xzy : c}(\to E)}{\lambda z^a.xzy : (a \to c)}(\to I)}{\lambda y^b.\lambda z^a.xzy : (b \to (a \to c))}(\to I)}{\lambda x^{a \to (b \to c)}.\lambda y^b.\lambda z^a.xzy : (a \to (b \to c)) \to (b \to (a \to c))}(\to I)$$

Type:

$$(a \to (b \to c)) \to (b \to (a \to c))$$

Programs & Types

Example 2

Program:

$$\dfrac{\dfrac{\dfrac{x : a \times b}{\overline{second}\, x : b}\,(\times E) \qquad \dfrac{x : a \times b}{\overline{first}\, x : a}\,(\times E)}{\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : b \times a}\,(\times I)}{\lambda x^{a \times b}.\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : (a \times b) \to (b \times a)}\,(\to I)$$

Type:

$$(a \times b) \to (b \times a)$$

Programs & Types

Example 3

Program:

$$\dfrac{\dfrac{p : a + (a \times b) \qquad \begin{array}{c}[x : a] \\ x : a\end{array} \qquad \dfrac{[y : a \times b]}{\overline{first}\, y : a}\,(\times E)}{\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y) : a}\,(+E)}{\lambda p^{a+(a\times b)}.(\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y)) : (a + (a \times b)) \to a}\,(\to I)$$

Type:

$$(a + (a \times b)) \to a$$

Programs & Types

Example 4

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to b \qquad y : a}{xy : b}(\to E) \qquad z : \neg b}{z(xy) : \bot}(\neg E)}{\lambda y^a.z(xy) : \neg a}(\to I)}{\lambda z^{\neg b}.\lambda y^a.z(xy) : \neg b \to \neg a}(\to I)}{\lambda x^{a \to b}.\lambda z^{\neg b}.\lambda y^a.z(xy) : (a \to b) \to (\neg b \to \neg a)}(\to I)$$

Type:

$$(a \to b) \to (\neg b \to \neg a)$$

Programs & Types

Example 5

Program:

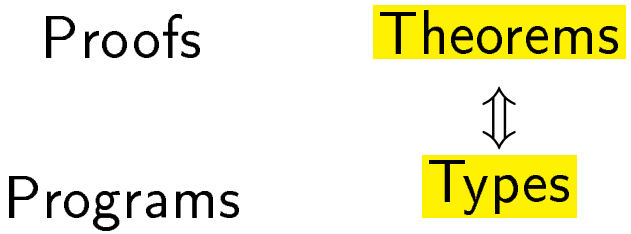
$$\dfrac{\dfrac{\dfrac{\dfrac{t : D \qquad r : \forall x.R(x,x)}{rt : R(t,t)}(\forall E) \qquad t : D}{\varepsilon y.(ry,t) : \exists y.R(t,y)}(\exists I)}{\lambda t.\varepsilon y.(ry,t) : \forall t.\exists y.R(t,y)}(\forall I)}{\lambda r.\lambda t.\varepsilon y.(ry,t) : \forall x.R(x,x) \to \forall t.\exists y.R(t,y)}(\to I)$$

Type:

$$\forall x.R(x,x) \to \forall x.\exists y.R(x,y)$$

Theorems & Types

Next, it is easy to observe that:



Types (specifications) and Theorems (propositions) share the same syntactic structure.

Theorems & Types

Example 1

Type or theorem?

Type:

$$(a \to (b \to c)) \to (b \to (a \to c))$$

Theorem:

$$(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$$

Theorems & Types

Example 2

Type or theorem?

Type:

$$(a \times b) \to (b \times a)$$

Theorem:

$$(a \wedge b) \Rightarrow (b \wedge a)$$

Theorems & Types

Example 3

Type or theorem?

Type:

$$(a + (a \times b)) \to a$$

Theorem:

$$(a \vee (a \wedge b)) \Rightarrow a$$

Theorems & Types

Example 4

Type or theorem?

Type:

$$(a \to b) \to (\neg b \to \neg a)$$

Theorem:

$$(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)$$

Theorems & Types

Example 5

Type or theorem?

Type:

$$\forall x.R(x,x) \to \forall x.\exists y.R(x,y)$$

Theorem:

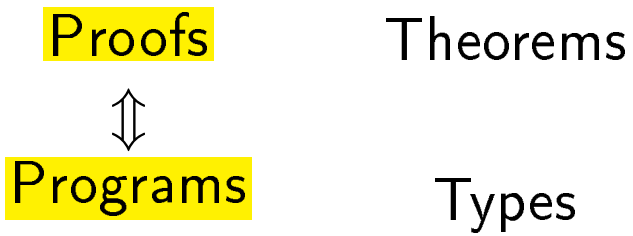
$$\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$$

The Isomorphism

Logic	Typed lambda calculus
$\Rightarrow$ (implication)	$\rightarrow$ (function type)
$\wedge$ (and)	$\times$ (product type)
$\vee$ (or)	$+$ (sum type)
$\forall$ (forall)	Π (pi type)
$\exists$ (exists)	Σ (sigma type)
$\top$	unit type
$\bot$	bottom type

Proofs & Programs

Finally, the isomorphism extends to:



One can be obtained directly from the other:

- From Program to Proof: by eliminating the terms and keeping only the types;
- From Proof to Program: by adding the terms with the corresponding types.

Proofs & Programs

Example 1

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow (b \Rightarrow c) \qquad a}{b \Rightarrow c}(\Rightarrow E) \qquad b}{c}(\Rightarrow E)}{a \Rightarrow c}(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}(\Rightarrow I)$$

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to (b \to c) \qquad z : a}{xz : b \to c}(\to E) \qquad y : b}{xzy : c}(\to E)}{\lambda z^a.xzy : (a \to c)}(\to I)}{\lambda y^b.\lambda z^a.xzy : (b \to (a \to c))}(\to I)}{\lambda x^{a \to (b \to c)}.\lambda y^b.\lambda z^a.xzy : (a \to (b \to c)) \to (b \to (a \to c))}(\to I)$$

Proofs & Programs

Example 2

Proof:

$$\dfrac{\dfrac{\dfrac{a \wedge b}{b}\,(\wedge E) \qquad \dfrac{a \wedge b}{a}\,(\wedge E)}{b \wedge a}\,(\wedge I)}{(a \wedge b) \Rightarrow (b \wedge a)}\,(\Rightarrow I)$$

Program:

$$\dfrac{\dfrac{\dfrac{x : a \times b}{\overline{second}\, x : b}\,(\times E) \qquad \dfrac{x : a \times b}{\overline{first}\, x : a}\,(\times E)}{\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : b \times a}\,(\times I)}{\lambda x^{a \times b}.\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : (a \times b) \to (b \times a)}\,(\to I)$$

Proofs & Programs

Example 3

Proof:

$$\dfrac{\dfrac{a \vee (a \wedge b) \qquad \begin{array}{c}[a] \\ a\end{array} \qquad \dfrac{[a \wedge b]}{a}\,(\wedge E)}{a}\,(\vee E)}{(a \vee (a \wedge b)) \Rightarrow a}\,(\Rightarrow I)$$

Program:

$$\dfrac{\dfrac{p : a + (a \times b) \qquad \begin{array}{c}[x : a] \\ x : a\end{array} \qquad \dfrac{[y : a \times b]}{\overline{first}\, y : a}\,(\times E)}{\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y) : a}\,(+E)}{\lambda p^{a+(a\times b)}.(\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y)) : (a + (a \times b)) \to a}\,(\to I)$$

Proofs & Programs

Example 4

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow b \qquad a}{b}(\Rightarrow E) \qquad \neg b}{\bot}(\neg E)}{\neg a}(\Rightarrow I)}{\neg b \Rightarrow \neg a}(\Rightarrow I)}{(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)}(\Rightarrow I)$$

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to b \qquad y : a}{xy : b}(\to E) \qquad z : \neg b}{z(xy) : \bot}(\neg E)}{\lambda y^{a}.z(xy) : \neg a}(\to I)}{\lambda z^{\neg b}.\lambda y^{a}.z(xy) : \neg b \to \neg a}(\to I)}{\lambda x^{a \to b}.\lambda z^{\neg b}.\lambda y^{a}.z(xy) : (a \to b) \to (\neg b \to \neg a)}(\to I)$$

Proofs & Programs

Example 5

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x.R(x,x)}{R(x,x)}(\forall E)}{\exists y.R(x,y)}(\exists I)}{\forall x.\exists y.R(x,y)}(\forall I)}{(\forall x.R(x,x) \Rightarrow (\forall x.\exists y.R(x,y))}(\Rightarrow I)$$

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{t : D \qquad r : \forall x.R(x,x)}{rt : R(t,t)}(\forall E) \qquad t : D}{\varepsilon y.(ry,t) : \exists y.R(t,y)}(\exists I)}{\lambda t.\varepsilon y.(ry,t) : \forall t.\exists y.R(t,y)}(\forall I)}{\lambda r.\lambda t.\varepsilon y.(ry,t) : \forall x.R(x,x) \to \forall t.\exists y.R(t,y)}(\to I)$$

Consequences

- To build a program that satisfies a specification (type):
  - Interpret the specification as a theorem (proposition);
  - Build a proof tree for this theorem;
  - Add terms with the corresponding types.
- To build a proof of a theorem:
  - Interpret the theorem as a specification;
  - Build a program that meets the specification;
  - Remove the terms from the derivation tree.

Consequences

Summary:

- To build a program is the same as to build a proof;
- To build a proof is the same as to build a program;
- To verify a program is the same as to verify a proof;
- Both verifications can be done via simple and efficient type checking algorithms.

Definition

A *Type Theory* is a theory that allows one to assign types to variables and construct complex type expressions. Then, lambda expressions can be derived to meet a certain type, or the type of an existing expression can be obained by following the theory's inference rules.

- Originally developed by Bertrand Russell in the 1910s as a tentative of fixing the paradoxes of set theory ("is the set composed of all sets that are not members of themselves a member of itself?");
- The *Simply Typed Lambda Calculus* is a type theory with a single operator ($\rightarrow$) and was developed by Church in the 1940s as a tentative of fixing the inconsistencies of the untyped lambda calculus;
- Since then it has been extended with many new operators;
- Various different type theories exist nowadays;
- *Martin Löf's Intuitionistic Type Theory* is one of the most important.

Constructivism and BHK

- Every true proposition must be accompanied by a proof of the validity of the statement; the proof must explain how to build the object that validates the argument (proposition);
- Proposed by Brouwer, Heyting and Kolgomorov, the BHK interpretation leaves behind the idea of the truth values of Tarski;
- $x : \sigma$ is interpreted as x *is a proof of* σ;

Constructivism and BHK

A proof of...

- $a \Rightarrow b$ is a mapping that creates a proof of b from a proof of a (*function*);
- $a \wedge b$ is a proof of a together with a proof of b (*pair*);
- $a \vee b$ is a proof of a or a proof of b together with an indication of the source (*pair*);
- $\forall x : A.P(x)$ is a mapping that creates a proof of $P(t)$ for every t in A (*function*);
- $\exists x : A.P(x)$ is an object t in A together with a proof of $P(t)$ (*pair*).

Constructivism and BHK

- Constructivism does not use the Law of the Excluded Middle $(p \vee \neg p)$ or any of its equivalents, that belong to classic logic only:
  - Double negation $\neg(\neg p) \Rightarrow p$;
  - Proof by contradiction $(\neg a \Rightarrow b) \wedge (\neg a \Rightarrow \neg b) \Rightarrow a$;
  - Peirce's Law $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$.
- A constructive proof is said to have *computational content*, as it is possible to "construct" the object that validates the proposition (the proof is a recipe for building this object);
- A constructive proof enables (computer) code *extraction* from proofs, thus the interest for it in computer science.

Constructivism

According to Troelstra:

> *"... the insistence that mathematical objects are to be constructed (mental constructions) or computed; thus theorems asserting the existence of certain objects should by their proofs give us the means of constructing objects whose existence is being asserted."*

Martin Löf's Intuitionistic Type Theory

A constructive type teory based on:

1. First-order logic to represent types and propositions;
2. Typed lambda calculus to represent programs and theorems.

and structured around the Curry-Howard Isomorphism.

- It is a powerful theory for sotware development and interactive theorem proving;
- Also used as a theory for the foundations of mathematics.

Inference rules for implication ($\Rightarrow$)

Introduction:

$$\frac{\begin{array}{c}[x:a]\\ \dots \\ y:b\end{array}}{\lambda x^a.y : a \Rightarrow b}\ (\Rightarrow I)$$

Elimination:

$$\frac{\lambda x^a.y : a \Rightarrow b \qquad z : a}{[z/x](\lambda x^a.y) : b}\ (\Rightarrow E)$$

Example 1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \Rightarrow (b \Rightarrow c) \qquad z : a}{xz : b \Rightarrow c}(\Rightarrow E) \qquad y : b}{xzy : c}(\Rightarrow E)}{\lambda z^{a}.xzy : (a \Rightarrow c)}(\Rightarrow I)}{\lambda y^{b}.\lambda z^{a}.xzy : (b \Rightarrow (a \Rightarrow c))}(\Rightarrow I)}{\lambda x^{a \Rightarrow (b \Rightarrow c)}.\lambda y^{b}.\lambda z^{a}.xzy : (a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}(\Rightarrow I)$$

Inference rules for conjunction ($\wedge$)

Introduction:

$$\frac{x : a \qquad y : b}{\overline{conj}\, x\, y : a \wedge b}\ (\wedge I)$$

Elimination 1:

$$\frac{z : a \wedge b}{\overline{first}\, z : a}\ (\wedge E_1)$$

Elimination 2:

$$\frac{z : a \wedge b}{\overline{second}\, z : b}\ (\wedge E_2)$$

Example 2

$$\dfrac{\dfrac{\dfrac{x : a \wedge b}{\overline{second}\, x : b}\,(\wedge E_2) \qquad \dfrac{x : a \wedge b}{\overline{first}\, x : a}\,(\wedge E_1)}{\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : b \wedge a}\,(\wedge I)}{\lambda x^{a \wedge b}.\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : (a \wedge b) \Rightarrow (b \wedge a)}\,(\Rightarrow I)$$

Inference rules for disjunction ($\vee$)

Introduction 1:

$$\frac{x : a}{\overline{inl}\, x : a \vee b}\ (\vee I_1)$$

Introduction 2:

$$\frac{y : b}{\overline{inr}\, y : a \vee b}\ (\vee I_2)$$

Inference rules for disjunction ($\vee$)

Elimination:

$$\frac{\begin{array}{ccc} & \begin{array}{c}[y:a]\\ \dots\\ p:c\end{array} & \begin{array}{c}[z:b]\\ \dots\\ q:c\end{array} \\ x:a\vee b & & \end{array}}{\overline{case}\,x(\lambda y.p)(\lambda z.q):c}\,(\vee E)$$

Example 3

$$\dfrac{\dfrac{p : a \vee (a \wedge b) \qquad \begin{array}{c}[x : a] \\ x : a\end{array} \qquad \dfrac{[y : a \wedge b]}{\overline{first}\, y : a}\,(\wedge E_1)}{\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y) : a}\,(\vee E)}{\lambda p^{a \vee (a \wedge b)}.(\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y)) : (a \vee (a \wedge b)) \Rightarrow a}\,(\Rightarrow I)$$

Inference rules for false ($\bot$)

Introduction:

No rule.

Elimination (*ex falso quodlibet*):

$$\frac{x : \bot}{\lambda \bot . x^{\bot} : P}\ (\bot E)$$

Inference rules for negation ($\neg$)

Introduction (same as implication introduction):

$$\frac{\begin{array}{c} x : A \\ \dots \\ f : \bot \end{array}}{\lambda x^A.f : \neg A} (\neg I, \text{ same as } \Rightarrow I)$$

Elimination (same as implication elimination):

$$\frac{x : A \qquad y : \neg A}{yx : \bot} (\neg E, \text{ same as } \Rightarrow E)$$

Example 4

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \Rightarrow b \qquad y : a}{xy : b}(\Rightarrow E) \qquad z : \neg b}{z(xy) : \bot}(\Rightarrow E)}{\lambda y^{a}.z(xy) : \neg a}(\Rightarrow I)}{\lambda z^{\neg b}.\lambda y^{a}.z(xy) : \neg b \Rightarrow \neg a}(\Rightarrow I)}{\lambda x^{a \Rightarrow b}.\lambda z^{\neg b}.\lambda y^{a}.z(xy) : (a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)}(\Rightarrow I)$$

Inference rules for universal quantifier ($\forall$)

Introduction:

$$\frac{\begin{array}{c}[x : A] \\ \dots \\ p : P(x)\end{array}}{\lambda x^A.p : \forall x.P(x)}\ (\forall I)$$

Elimination:

$$\frac{a : A \qquad f : \forall x.P(x)}{fa : [a/x]P}\ (\forall E)$$

Inference rules for existential quantifier ($\exists$)

Introduction:

$$\frac{a : D \qquad f(a) : P(a)}{\varepsilon x.(f(x), a)) : \exists x.P(x)}\ (\exists I)$$

Elimination:

$$\frac{r : \exists x.P(x) \qquad \begin{array}{c}[t : D, g(t) : P(t)] \\ \cdots \\ h(t, g) : C\end{array}}{E(r, \lambda g.\lambda t.h(t, g)) : C}\ (\exists E)$$

Example 5

$$\dfrac{\dfrac{\dfrac{\dfrac{t : D \qquad r : \forall x.R(x,x)}{rt : R(t,t)}\,(\forall E) \qquad t : D}{\varepsilon y.(ry,t) : \exists y.R(t,y)}\,(\exists I)}{\lambda t.\varepsilon y.(ry,t) : \forall t.\exists y.R(t,y)}\,(\forall I)}{\lambda r.\lambda t.\varepsilon y.(ry,t) : \forall x.R(x,x) \Rightarrow \forall t.\exists y.R(t,y)}\,(\Rightarrow I)$$

General

A richly typed lambda calculus extended with inductive definitions.

- *Calculus of Constructions* developed by Thierry Coquand;
- Constructive type theory;
- Later extended with inductive definitions;
- Used as the mathematical language of the Coq proof assistant

Calculus of Constructions

- All logical operators ($\rightarrow, \wedge, \vee, \neg$ and $\exists$) are defined in terms of the universal quantifier ($\forall$), using "dependent types";
- Types and programs (terms) have the same syntactical structure;
- Types have a type themselves (called "Sort");
- Base sorts are "$Prop$" (the type of propositions) and "Set" (the type of small sets);
- $Prop : Type(1)$, $Set : Type(1)$, $Type(i) : Type(i+1), i \geq 1$;
- $S = \{Prop, Set, Type(i) | i \geq 1\}$ is the set of sorts;
- Various datatypes can be defined (naturals, booleans etc);
- Set of typing and conversion rules.

Inductive Definitions

Finite definition of infinite sets.

- "Constructors" define the elements of a set;
- Constructors can be base elements of the set;
- Constructors can be a functions that takes set elements and return new set elements.
- Manipulation is done via "pattern matching" over the inductive definitions.

Inductive Definitions

Booleans

```
{false,true}

Inductive boolean:
 | false: boolean
 | true: boolean.

Variable x: boolean.
Definition f: boolean:= false.
```

Inductive Definitions

Naturals

```
{0, 1, 2, 3, ...} = {O, SO, SSO, SSSO, ...}

Inductive nat:=
 | O: nat
 | S: nat->nat.

Variable y: nat.
Definition zero: nat:= O.
Definition one: nat:= S O.
Definition two: nat:= S one.
```

Inductive Definitions

String sets

```
Inductive ss:=
 | ss_empty: ss
 | ss_item: string->ss
 | ss_build: string->ss->ss.

Variable z: ss.
Definition ss0: ss:= ss_empty.
Definition ss1: ss:= ss_item "abc".
Definition ss2: ss:= ss_build "def" (ss_item "abc").
Definition ss3: ss:= ss_build "ghi" (
                     ss_build "def" (ss_item "abc")).
```

Pattern matching

Booleans

```
Definition negb (x: bool): bool:=
match x with
 | false => true
 | true => false
end.
```

Pattern matching

Naturals

```
Definition sub (n: nat): nat :=
match n with
 | O => O
 | S m => m
end.

Fixpoint nat_equal (n1 n2: nat): bool :=
match n1, n2 with
 | O, O => true
 | S m, S n => nat_equal m n
 | O, S n => false
 | S m, O => false
end.
```

Characteristics

- Software tools that assist the user in theorem proving and program development;
- First initiative dates from 1967 (Automath, De Bruijn);
- Many provers are available today (Coq, Agda, Mizar, HOL, Isabelle, Matita, Nuprl...);
- Interactive;
- Graphical interface;
- Proof/program checking;
- Proof/program construction.

Usage

1. The user writes a theorem (proposition) in first-order logic or a type expression (specification);
2. The constructs directly or indirectly:
   - A proof of the theorem;
   - A program (term) that complies to the specification.
3. Directly: the proof/term is written in the formal language accepted by the assistant;
4. Indirectly: the proof/term is built with the assistance of an interactive "tactics" language:
5. In either case, the assistant checks that the proof/term complies to the theorem/specification.

Check and/or construct

- Proof assistants check that proofs/terms are correctly constructed;
- This is done via simple type-checking algorithms;
- Automated proof/term construction might exist is some cases, to some extent, but is not the main focus;
- Thus the name “proof assistant”;
- Automated theorem proofing might be pursued, due to “proof irrelevance”;
- Automated program development, on the other hand, is unrealistic.

Main benefits

- Proofs and programs can be mechanically checked, saving time and effort and increasing reliability;
- Checking is efficient;
- Results can be easily stored and retrieved for use in different contexts;
- Tactics help the user to construct proofs/programs;
- User gets deeper insight into the nature of his proofs/programs, allowing further improvement.

Applications

- Formalization and verification of theorems and whole theories;
- Verification of computer programs;
- Correct software development;
- Automatic review of large and complex proofs submitted to journals;
- Verification of hardware and software components.

Drawbacks

- Failures in infrastructure may decrease confidence in the results (proof assistant code, language processors, operating system, hardware etc);
- Size of formal proofs;
- Reduced numer of people using proof assistants;
- Slowly increasing learning curve;
- Resemblance of computer code keeps pure mathematicians uninterested.

Overview

- Developed by Huet/Coquand at INRIA in 1984;
- First version released in 1989, inductive types were added in 1991;
- Continuous development and increasing usage since then;
- The underlying logic is the Calculus of Constructions with Inductive Definitions;
- It is implemented by a typed functional programming with a higher order logic language called *Gallina*;
- Interaction with the user is via a command language called *Vernacular*;
- Constructive logic with large standard library and user contributions base;
- Extensible environment;
- All resources freely available from `http://coq.inria.fr/`.

User session

The proof can be constructed directly ou indirectly.
In the indirect case,

- The initial goal is the theorem/specification supplied by the user;
- The environment and the context are initially empty;
- The application of a "tactics" substitutes the current goal for zero ou more subgoals;
- The context changes and might incorporate new hypotheses;
- The process is repeated for each subgoal, until no one subgoal remains;
- The proof/term is constructed from the sequence of tactics used.

Tactics usage

- Inference rules map premises to conclusions;
- *Forward reasoning* is the process of moving from premises to conclusions;
  - Example: from a proof of a and a proof of b one can prove $a \wedge b$;
- *Backward reasoning* is the process of moving from conclusions to premises;
  - Example: to prove $a \wedge b$ one has to prove a and also prove b;
- Coq uses *backward reasoning*;
- They are implemented by "tactics";
- A tactic reduces a goal to its subgoals, if any.

Example 1: Coq session

Direct proof construction

```
Parameters a b c: Prop.

Definition t0: (a->b->c)->b->a->c:=
  fun (H: a->b->c)(H1: b)(H2: a)=>
  H H2 H1.
```

Example 1: Coq session

Indirect proof construction

```
Parameters a b c: Prop.
Theorem t1: (a->(b->c))->(b->(a->c)).
Proof.
intro H.
intro H1.
intro H2.
apply H.
exact H2.
exact H1.
Qed.
```

Example 1: Coq proof

```
fun
(H : a -> b -> c)
(H1 : b)
(H2 : a)
=>
H H2 H1
: (a -> b -> c) -> b -> a -> c
```

$$\lambda x^{a \Rightarrow (b \Rightarrow c)}.\lambda y^{b}.\lambda z^{a}.xzy : (a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$$

Example 2: Coq session

Indirect proof construction

```
Parameters a b: Prop.
Theorem t2: (a /\ b)->(b /\ a).
Proof.
intro H.
destruct H as [H1 H2].
split.
exact H2.
exact H1.
Qed.
```

Example 2: Coq proof

```
fun
H : a /\ b
=>
match H with
| conj H1 H2 => conj H2 H1
end
: a /\ b -> b /\ a
```

$$\lambda x^{a \wedge b}.\overline{conj}(\overline{second}\, x)(\overline{first}\, x) : (a \wedge b) \Rightarrow (b \wedge a)$$

Example 3: Coq session

Indirect proof construction

```
Parameters a b: Prop.
Theorem t3: (a\/(a /\ b))->a.
Proof.
intro H.
destruct H as [H1 | H2].
trivial.
destruct H2 as [H3 H4].
exact H3.
Qed.
```

Example 3: Coq proof

```
fun
H : a \/ a /\ b
=>
match H with
| or_introl H1 => H1
| or_intror (conj H3 _) => H3
end
: a \/ a /\ b -> a
```

$$\lambda p^{a \vee (a \wedge b)}.(\overline{case}\, p\, (\lambda x.x)(\lambda y.\overline{first}\, y)) : (a \vee (a \wedge b)) \Rightarrow a$$

Example 4: Coq session

Indirect proof construction

```
Parameters a b: Prop.
Theorem t4: (a->b)->(~ b->~ a).
Proof.
intro H.
intro H1.
intro H2.
apply H1.
apply H.
exact H2.
Qed.
```

Example 4: Coq proof

```
fun
(H : a -> b)
(H1 : ~ b)
(H2 : a)
=>
H1 (H H2)
: (a -> b) -> ~ b -> ~ a
```

$$\lambda x^{a \to b}.\lambda z^{\neg b}.\lambda y^{a}.z(xy) : (a \to b) \to (\neg b \to \neg a)$$

Example 5: Coq session

Indirect proof construction

```
Parameter R: Prop->Prop->Prop.
Theorem t5: (forall x: Prop, R x x)->
            (forall x: Prop, exists y: Prop, R x y).
Proof.
intro H.
intro x.
exists x.
exact (H x).
Qed.
```

Example 5: Coq proof

```
fun
(H : forall x : Prop, R x x)
(x : Prop)
=>
ex_intro (fun y : Prop => R x y) x (H x)
: (forall x : Prop, R x x) ->
   forall x : Prop, exists y : Prop, R x y
```

$$\boxed{\lambda r.\lambda t.\varepsilon y.(ry, t) : (\forall x.R(x,x) \Rightarrow (\forall x.\exists y.R(x,y))}$$

Program specification

Sorting algorithm

How to specify a sorting algorithm?

- Define the domain (lists of natural numbers);
- Relate pieces of data:
  - Input: a list of naturals;
  - Output: a list of naturals;
  - Requirement 1: the lists must have the same elements (permutation);
  - Requirement 2: the output list must be "sorted"
- Write the proposition;
- Prove the theorem.

Program specification

Sorting algorithm

What is a sorted list?

```
Inductive sorted: list nat->Prop:=
   | sorted0: sorted (nil)
   | sorted1: forall n: nat, sorted (n::nil)
   | sorted2: forall n1 n2: nat, forall l: list nat,
     n1<=n2 -> sorted (n2::l) -> sorted (n1::n2::l).
```

Program specification

Sorting algorithm

What is a permutation of a list?

```
Fixpoint number_of_occur (n: nat)(l: list nat): nat:=
   match l with
   | nil => 0
   | cons n' l' =>if (beq_nat n n')
                  then S (number_of_occur n l')
                  else (number_of_occur n l')
   end.
Definition perm (l1 l2: list nat): Prop:= forall n: nat,
   number_of_occur n l1 = number_of_occur n l2.
```

Program specification

Sorting algorithm

The type of functions that sort lists:

```
Theorem sort:
        forall (l1: list nat),
        exists (l2: list nat),
        (perm l1 l2) /\ (sorted l2).
```

Certified software development

1. Write the specifications as type expressions;
2. Interpret them as theorems;
3. Build the proofs;
4. Let the proof assistant check them;
5. Convert them to computer programs or use the code extraction facility.

Introduction

- Great and increasing interest in formal proof and program development over the recent years;
- Main areas include:
  - Programming language semantics formalization;
  - Mathematics formalization;
  - Education.
- Important projects in both academy and industry;
- Top 100 theorems (88% formalized);
- The trend is clearly set.

Four Color Theorem

- Stated in 1852, proved in 1976 and again in 1995;
- The two proofs used computers to a some extent, but were not fully mechanized;
- In 2005, Georges Gonthier (Microsoft Research) and Benjamin Werner (INRIA) produced a proof script that was fully checked by a machine;
- Milestone in the history of computer assisted proofing;
- 60,000 lines of Coq script and 2,500 lemmas;
- Byproducts.

Four Color Theorem

"Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction using mathematics to help programming computers."

Georges Gonthier

Odd Order Theorem

- Also known as the Feit-Thomson Theorem;
- Important to mathematics (in the classification of finite groups) and cryptography;
- Conjectured in 1911, proved in 1963;
- Formally proved by a team led by Georges Gonthier in 2012;
- Six years with full-time dedication;
- Huge achievement in the history of computer assisted proofing;
- 150,000 lines of Coq script and 13,000 theorems;

Opportunity

Fermat's Last Theorem

Statement in Coq:

```
Theorem Fermat: forall x y z n: nat, (x^n+y^n=z^n)->(n<=2).
```

Compiler Certification

- CompCert, a fully verified compiler for a large subset of C that generates PowerPC code;
- Object code is certified to comply with the source code in all cases;
- Applications in avionics and critical software systems;
- Not only checked, but also developed in Coq;
- Three persons-years over a five yers period;
- 42,000 lines of Coq code.

Microkernel Certification

- Critical component of operating systems, runs in privileged mode;
- Harder to test in all situations;
- seL4, written in C (10,000 lines), was fully checked in HOL/Isabelle;
- No crash, no execution of any unsafe operation in any situation;
- Proof is 200,000 lines long;
- 11 persons-years, can go down to 8, 100% overhead over a non-certified project.

Digital Security Certification

- JavaCard smart card platform;
- Personal data such as banking, credit card, health etc;
- Multiple applications by different companies;
- Confidence and integrity must be assured;
- Formalization of the behaviour and the properties of its components;
- Complete certification, highest level achieved;
- INRIA, Schlumberger and Gemalto.

Origins

- Years of experience in teaching language and automata theory;
- Book "Linguagens Formais" published in 2009 (with J.J. Neto and I.S. Vega);
- Algorithms were used instead of demonstrations for most theorems;
- Interest in formalization after studying logic, lambda calculus, type theory and Coq;
- Desire to follow the lines of the book and formalize its contents;
- Related work over recent years, usually with a restricted focus.

Situation 1

- 6,000 lines of Coq script over a 5 months period;
- Representation of all relevant objects of the universe of discourse:
  - Symbols (including terminal and non-terminal);
  - Strings (including sentences and sentential forms);
  - String sets;
  - Regular sets (including regular expressions);
  - Grammars (of type 3, 2, 1 and 0);
  - Finite automata (including non-deterministic and automata with empty transitions);
  - Stack automata (with acceptance by final state and empty stack);
  - Turing Machines.
- Basically via inductive types and predicates, definitions and fixpoints.

Situation 2

- Many algorithms were implemented;
- They operate on the objects and resemble functional programs;
- Grouped in a few libraries;
- However, the correctness of these algorithms must be proved;
- To achieve this, their semantics must be formalized in the form of properties described by propositions;
- More general lemmas and theorems must also be stated and proved;
- Not much has been done in this area so far.

Libraries

Library `Regular_expressions`

1. Convert to regular set (and back);
2. Generate all sentences up to n derivations.

Libraries

Library `Grammars`

1. Check whether a grammar is type 3 (right or left linear), type 2, type 1 or type 0;
2. Check whether a grammar is in a normal formal, either Chomsky or Greibach;
3. Generate all sentential forms up to n derivations;
4. Generate all sentences up to n derivations.

Libraries

Library `Finite_automata`

1. Check whether a finite automaton is free of empty transitions;
2. Check whether a finite automaton is deterministic;
3. Check whether a finite automaton has a total transition function;
4. Generate all configurations up to n movements;
5. Generate all sentences up to n movements.

Libraries

Library `Stack_automata`

1. Check whether a stack automaton is deterministic;
2. Generate all configurations up to n movements;
3. Generate all sentences up to n movements.

Libraries

Library `Turing_machine`

1. Check whether a Turing machine is deterministic;
2. Generate all configurations up to n movements;
3. Generate all sentences up to n movements.

Libraries

Library `Equivalences_type_3`

1. Check the equivalence of the language accepted by two finite automata;
2. Check the equivalence of the language accepted by a finite automaton and a regular expression.
3. Check the equivalence of the language accepted by a type 3 grammar and a regular expression.

Libraries

Library `Conversions_type_3`

1. Remove unreachable states from a finite automaton;
2. Remove useless states from a finite automaton;
3. Remove empty transitions from a finite automaton;
4. Remove non determinism from a finite automaton;
5. Convert the transition function of a finite automaton to a total one;
6. Minimize the number of states in a finite automaton.

Libraries

Library `Operations_type_3`

1. Concatenation of two finite automata;
2. Union of two finite automata;
3. Closure of a finite automaton;
4. Complement of a finite automaton;
5. Intersection of two finite automata.

Libraries

Library `Equivalences_type_3`

1. Check the equivalence of two different type 3 grammars;
2. Check the equivalence of two different regular expressions;
3. Check the equivalence of two different finite automata;
4. Convert a regular expression into an equivalent finite automaton;
5. Convert a type 3 grammar into an equivalent finite automaton;
6. Convert a regular expression into an equivalent type 3 grammar;

Libraries

Library `Decidable_questions_type_3`

1. Check whether a string is accepted by a finite automaton;
2. Check whether a string is generated by a regular expressions;
3. Check whether the language accepted by a finite automaton is empty.

Objectives 1

Prove theorems about:

- Closure properties for different language classes;
- Decidable properties for different language classes;
- Equivalence between regular expressions, deterministic/non-deterministic/empty transitions finite automata and left/right linear grammars;
- Existence and uniqueness of minimal finite automata;
- Equivalence between stack automata and context-free grammars;
- Equivalence between the different acceptance criteria for stack automata;
- Existence of normal forms for context-free grammars;
- Non-equivalence between deterministic and non-deterministic stack automata;

Objectives 2

Prove theorems about:

- Pumping lemma for regular and context-free languages;
- Equivalence between context-sensitive grammars and linear-bounded automata;
- Existence of normal forms for context-sensitive grammars;
- Equivalence between unrestricted grammars and deterministic/non-deterministic Turing machines;
- Existence of non-context-sensitive, non-recursive and non-recursively enumerable languages.

Last but not least:

- Use Coq's code extraction facility to obtain certified programs that manipulate the objects of language and automata theory.

Example 1

Context-sensitive grammar

The context sensitive language $\{a^n b^n c^n, n \geq 1\}$ is generated by the following grammar:

$$
\begin{aligned}
X &\rightarrow aXBC \\
X &\rightarrow abC \\
CB &\rightarrow BC \\
bB &\rightarrow bb \\
bC &\rightarrow bc \\
cC &\rightarrow cc
\end{aligned}
$$

Example 1

Context-sensitive grammar

Definitions of “rule”, “rule set” and “grammar”:

```
Definition rule: Type := (string * string)%type.
Inductive rule_set: Type :=
  | rset_item: rule->rule_set
  | rset_build: rule->rule_set->rule_set.
Definition grammar: Type := (string * rule_set)%type.
```

Example 1

Context-sensitive grammar

The rules of this grammar can then be defined according the Coq definitions as:

```
Definition r0: rule := ("X","aXBC").
Definition r1: rule := ("X","abC").
Definition r2: rule := ("CB","BC").
Definition r3: rule := ("bB","bb").
Definition r4: rule := ("bC","bc").
Definition r5: rule := ("cC","cc").
```

Example 1

Context-sensitive grammar

The rule set and the grammar itself can then be defined according the Coq definitions as:

```
Definition rs0: rule_set :=
rset_build r0 (
rset_build r1 (
rset_build r2 (
rset_build r3 (
rset_build r4 (
(rset_item r5))))))).
Definition g0: grammar := ("X",rs0).
```

Example 1

Context-sensitive grammar

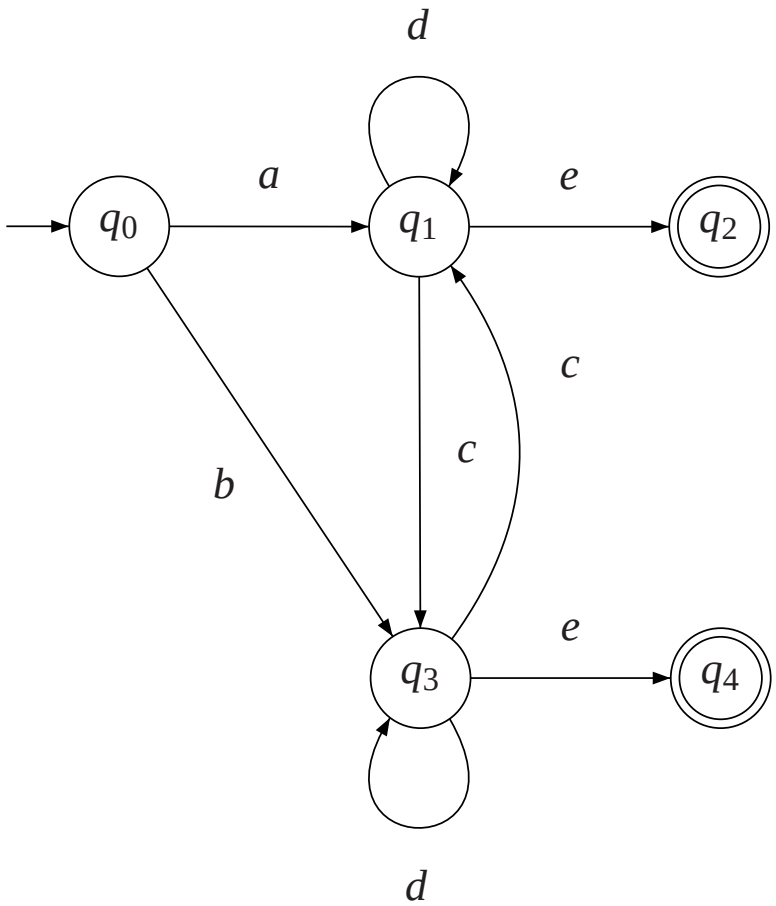
As an example, the set of all sentential forms produced after four derivation steps with the example grammar is displayed by Coq as:

```
: 0
X
: 1
aXBC, abC,
: 2
aaXBCBC, aabCBC, abc,
: 3
aaaXBCBCBC, aaabCBCBC, aaXBBCC, aabBCC, aabcBC,
: 4
aaaaXBCBCBCBC, aaaabCBCBCBC, aaaXBBCCBC, aaabBCCBC,
aaabcBCBC, aaaXBCBBCC, aaabCBBCC, aabbCC,
```

Example 2

Finite automaton

Finite automaton for $(a|b)(c|d)^*e$:



Example 2

Finite automaton

Definitions of “state”, “input”, “transition”, “transition set” and “finite automata”:

```
Definition state: Type := (nat*bool*bool)%type.
Definition input: Type := option ascii.
Definition trans: Type := (state*input*state)%type.
Inductive ts: Type :=
    | ts_empty: ts
    | ts_item: trans->ts
    | ts_build: trans->ts->ts.
Definition f_automata: Type := (state*ts)%type.
```

Example 2

Finite automaton

Description of the states of the automaton:

```
Definition s0: state := (0,true,false).
Definition s1: state := (1,false,false).
Definition s2: state := (2,false,true).
Definition s3: state := (3,false,false).
Definition s4: state := (4,false,true).
```

Example 2

Finite automaton

Description of the transitions of the automaton:

```
Definition t0: trans := (s0,Some "a"%char,s1).
Definition t1: trans := (s0,Some "b"%char,s3).
Definition t2: trans := (s1,Some "c"%char,s3).
Definition t3: trans := (s1,Some "d"%char,s1).
Definition t4: trans := (s1,Some "e"%char,s2).
Definition t5: trans := (s3,Some "c"%char,s1).
Definition t6: trans := (s3,Some "d"%char,s3).
Definition t7: trans := (s3,Some "e"%char,s4).
```

Example 2

Finite automaton

Description of the transition set and the automaton itself:

```
Definition ts0: ts :=
ts_build t0 (
ts_build t1 (
ts_build t2 (
ts_build t3 (
ts_build t4 (
ts_build t5 (
ts_build t6 (
ts_item t7))))))).
Definition a0: f_automata := (s0,ts0).
```

Example 2

Finite automaton

Original automaton:

```
Initial state: q0
Input symbols: a b c d e
States:
q0 - Initial
q1
q2 - Final
q3
q4 - Final
Rules:
(q0,a)->(q1)
(q0,b)->(q3)
(q1,c)->(q3)
(q1,d)->(q1)
(q1,e)->(q2)
(q3,d)->(q3)
(q3,e)->(q4)
(q3,c)->(q1)
```

Example 2

Finite automaton

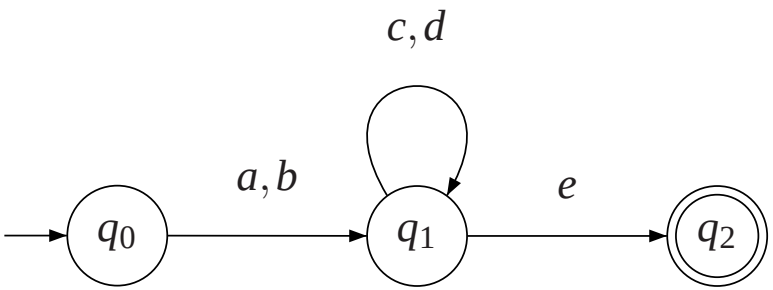
After executing the minimization algorithm:

```
Initial state: q0
Input symbols: a b c d e
States:
q0 - Initial
q1
q2 - Final
Rules:
(q0,a)->(q1)
(q0,b)->(q1)
(q1,c)->(q1)
(q1,d)->(q1)
(q1,e)->(q2)
```

Example 2

Finite automaton

Minimized finite automaton for $(a|b)(c|d)^*e$:



Next steps

- Although developed on Coq, this is just functional programming;
- It is necessary to certify these algorithms;
- For that, one must first describe its semantics;
- This is done via lemmas that related pieces of data;
- An example follows;
- These theorems must be proved.

Next steps

```
Definition fa_cat (a1 a2: f_automata): f_automata := ...
Lemma fa_cat_str:
      forall s1 s2: string,
      forall a1 a2: f_automata,
      (fa_belongs_p s1 a1) /\ (fa_belongs_p s2 a2) ->
      (fa_belongs_p (s1++s2) (fa_cat a1 a2)).
Proof.
...
Lemma fa_cat_str_conv:
      forall (s: string) (a1 a2: f_automata),
      fa_belongs_p s (fa_cat a1 a2) -> exists (s1 s2: string),
      s = s1 ++ s2 /\ fa_belongs_p s1 a1 /\ fa_belongs_p s2 a2.
Proof.
...
```

Next steps

Another approach is to use the code extraction facility in Coq:

- Computer code (programs) can be extracted directly from proofs;
- Constructive proofs show how to build the object that validates the proposition.

Next steps

```
Lemma fa_cat_str_2:
      forall s1 s2: string,
      forall a1 a2: f_automata,
      (fa_belongs_p s1 a1) -> (fa_belongs_p s2 a2) ->
      exists a: f_automata, fa_belongs_p (s1++s2) a.
Proof.
...
Lemma fa_cat_str_conv_2:
      forall (s: string) (a1 a2: f_automata),
      exists a: f_automata,
      fa_belongs_p s a -> exists (s1 s2: string),
  s = s1 ++ s2 / fa_belongs_p s1 a1 / fa_belongs_p s2 a2.
Proof.
...
```

Next steps

Of course, all other lemmas and theorems...

```
Theorem pl_reg:
   forall l: ss,
   is_type3_l l ->
   exists n: nat, forall s: string, exists x y z: string,
   in_set_p s l /\
   length s>=n /\
   str_equal s (x++y++z) ->
   length y>=1 /\
   length (x++y)<=n /\
   forall i: nat, in_set_p (x ++ (str_n y n) ++ z) l.
```

Applications

- Academy;
- Industry;
- Software and hardware certification;
- Software and hardware development;
- Proof checking;
- Theoretical formalization;
- Mathematics database (e.g. QED project).

Adaptive Technology

Plenty of results and artifacts for:

- Modelling;
- Representation;
- Simulation;
- Translation.

Not much has been done in:

- Discovering, stating and proving properties (equivalences, closure, decidability, classification etc);
- Theoretical development and theorem proving.

$\Rightarrow$ A proof assistant such as Coq can be the ideal environment for pursuing these objectives.

Computers and mathematics

- Practitioners base is still small;
- Learning curve grows slowly;
- Advantages of formalization are immense;
- Important industrial projects;
- Disadvantages are being gradually eliminated.

Computers and mathematics

- Not easy, but very rewarding;
- Hope you have enjoyed;
- Hope you want to go further;
- Ask me if you want references;
- Write me if you have questions or suggestions;
- Let me know you if plan to work in this area;
- Hope to bring more next time;

Thank you!