Extraction of Programs from Proofs

Rafael Castro

rafaelcgs10@gmail.com

Departamento de Ciência da Computação
Centro de Ciências e Tecnológicas
Universidade do Estado de Santa Catarina

07/03/2018

Sistemas de Provas

- Sistemas/Cálculos de provas servem para construir provas de uma maneira muito formal.
- São uma coleção de regras que explicam como derivar novas fórmulas.
- Um sistema de prova pode ser utilizado na formalização de diversas lógicas, como Lógica Proposicional e Lógica de Predicados.
- Os principais sistemas de provas são a Dedução Natural e o Cálculo de Hilbert.

Fragmento Proposicional Clássico da Dedução Natural

$$\frac{}{u : A}\ (hip)$$

$$\frac{A \to B \qquad A}{B}\ (\to^-) \qquad\qquad \frac{B}{A \to B}\ (\to^+)\ [u : A]$$

$$\frac{A \wedge B}{A}\ (\wedge_l^-) \qquad\qquad \frac{A \wedge B}{B}\ (\wedge_r^-) \qquad\qquad \frac{A \qquad B}{A \wedge B}\ (\wedge^+)$$

$$\frac{A \vee B \qquad A \to C \qquad B \to C}{C}\ (\vee^-)$$

$$\frac{A}{A \vee B}\ (\vee_l^+) \qquad\qquad \frac{B}{A \vee B}\ (\vee_r^+)$$

$$\frac{\bot}{A}\ (efq) \qquad\qquad \frac{\neg\neg A}{A}\ (raa)$$

onde $\neg A \Rightarrow A \to \bot$.

Exemplo de Prova em DN

$$
\dfrac{\dfrac{\dfrac{}{u : (A \lor \neg A) \to \bot}\,(\text{hip}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{u : (A \lor \neg A) \to \bot}\,(\text{hip}) \quad \dfrac{\dfrac{}{v : A}\,(\text{hip})}{A \lor \neg A}\,(\lor_r^+)}{\bot}}{A \to \bot}\,(\to^+)[v]}{(A \lor \neg A)}\,(\lor_l^+)}{\bot}\,(\to^-)}{((A \lor \neg A) \to \bot) \to \bot}\,(\to^+)[u]
$$

$$
\dfrac{((A \lor \neg A) \to \bot) \to \bot}{A \lor \neg A}\,(raa)
$$

Cálculo Lambda

- O Cálculo Lambda é um modelo de computação criado por Alonzo Church em 1933.
- O proposito inicial do Cálculo Lambda foi ser uma linguagem de macros para um lógica e assim demonstrar a indecibilidade do problema da parada.
- Funciona como um sistema de reescrita: existem regras para reescrever expressões.
- A primeira linguagem de programação. Uma década antes do primeiro computador.

Sintaxe:

$$e \ := (e\ e') \mid (\lambda x.e) \mid x$$

Reescrita:

$$(\lambda x.e)\ e' \Rightarrow_\beta [e'/x]e$$

Exemplos de Computação em CL

1)

$$(\lambda x.x)\ y \Rightarrow_\beta y$$

2)

$$(\lambda x.x)\ (\lambda x.x) \Rightarrow_\beta \lambda x.x$$

3)

$$((\lambda x.\lambda y.y)\ a)b \Rightarrow_\beta b$$

4)

$$(\lambda x.xx)\ (\lambda x.xx) \Rightarrow_\beta (\lambda x.xx)\ (\lambda x.xx)$$

Cálculo Lambda Tipado

O Cálculo Lambda é muito poderoso, permite criar o Rightarrowalente de fórmulas lógicas infinitas e assim a lógica representada é inconsistente.
Para evitar paradoxos Church utilizou o mesmo truque que Bertrand Russel: Type Thoery.

Cálculo Lambda Tipado

$$\frac{}{x : A \vdash x : A}\ (hip)$$

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{e\ e' : \Gamma \vdash B}\ (\to^{-}) \quad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\ (\to^{+})$$

$$\frac{\Gamma \vdash e : A * B}{\Gamma \vdash fst\ e : A}\ (fst) \quad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash snd\ e : B}\ (snd)$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e' : B}{\Gamma \vdash (e, e') : A \wedge B}\ (pair)$$

$$fst(e, e') \Rightarrow_{\beta} e$$
$$snd(e, e') \Rightarrow_{\beta} e'$$

Cálculo Lambda Tipado

$$\frac{\Gamma \vdash e : A + B \qquad \Gamma \vdash e' : A \to C \qquad \Gamma \vdash e'' : B \to C}{\Gamma \vdash case(e, e', e'') : C} \; (+^-)$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (0, e) : A + B} +_l^+ \qquad \frac{B}{\Gamma \vdash (1, e) : A + B} +_r^+$$

$$case((0, e), e', e'') \Rightarrow_\beta e' \; e$$
$$case((1, e), e', e'') \Rightarrow_\beta e'' \; e$$

Isomorfismo de Curry-Howard

O Isomorfismo de Curry-Howard é uma observação que provas construtivas na dedução estão numa correspondência natural com programas em Cálculo Lambda.

$$\frac{A \to B \qquad A}{B}\,(\to^-) \qquad\qquad \frac{B}{A \to B}\,(\to^+)\ [u : A]$$

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{e\ e' : \Gamma \vdash B}\,(\to^-) \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\,(\to^+)$$

Realizability

Realizability

Extracting a Logarithm Approximation Algorithm

$$\forall n, \exists k\ r, 0 < n \to n = 2^k + r \land r < 2^k$$

The intuition here is that we should increase r when $r < 2^k$ or increase k otherwise.

Sketch of the proof

Induction over n:

1. Base case: $n = 0$. So since we have $0 < 0$, we just use *efq*.
2. Step case: $n = Succ\ m$.
    1. The base case of induction is not the base of the recursion in the algorithm, because $log_2\ 0 = -\infty$.
    2. We do case analysis on $0 < m$, so we have $0 < m$ or $m = 0$.
    3. This way we have two new goals on the induction step:
        1. if $m = 0$, then $n = 1$ (our base case of recursion) and we shall prove:

           $$\exists k\ r, 1 = 2^k + r \wedge r < 2^k$$

           So $k = r = 0$.
        2. if $0 < m$, then the recursion can go on and we should prove that:

           $$\exists k\ r, Succ\ m = 2^k + r \wedge r < 2^k$$

Sketch of the proof

Our current goal:

$$\exists k\ r, Succ\ m = 2^k + r \wedge r < 2^k$$

We need to do case analysis on $Succ\ r < 2^k$ because that is rule for increasing k or r.

1. If $Succ\ r < 2^k$, then we chose $k = k$ and $r = Succ\ r'$.
2. If $(Succ\ r < 2^k) \rightarrow \bot$, then we chose $k = Succ\ k'$ and $r = 0$.