

# Uso de Coq para verificação de propriedades de sistemas de tipos

Rafael Castro

[rafaelcgs10@gmail.com](mailto:rafaelcgs10@gmail.com)

Departamento de Ciéncia da Computaçao  
Centro de Ciéncias e Tecnológicas  
Universidade do Estado de Santa Catarina

22 de Junho de 2018

# O que são assistentes de provas?

- Assistentes de provas ou provadores interativos são programas para o desenvolvimento de provas formais.
- O núcleo de um assistente de provas é um verificador, que verifica a consistência lógica da prova.
- Fornecem de maneira interativa de visualizar as informações sobre o estado atual da prova.
- A verificação humana de provas é demorada e sujeita a falhas: Último Teorema de Fermat.  
Assistentes de provas permitem provar coisas que não seriam realizáveis somente com papel e caneta!

## O que é Coq?

- Coq é um assistentes de provas desenvolvido desde 1984 pelo French Institute for Research in Computer Science and Automation (INRIA).
- Coq é fruto de sistemas de tipos: baseado em **Higher order dependently typed polymorphic lambda calculus**, o nomeado Calculus of Constructions (CoC).
- Coq suporta programação com tipos dependentes e lógica de alta ordem.

# Objetivo do SEC

Introdução

A linguagem: STLC  
Semântica e Sistemas de Tipos  
Propriedades do Sistema de Tipos

- Formalizar uma linguagem de programação funcional em Coq.
  - Sintaxe: termos e tipos;
  - Semântica: *Small Step*;
  - Sistemas de tipos: Regras de inferência de tipo;
- Provar propriedades do sistema de tipos

# A linguagem: STLC

- Cálculo Lambda Simplesmente Tipado: *Simple Typed Lambda Calculus* (STLC).

$t ::= x$	variable
$\lambda x:T_1.t_2$	abstraction
$t_1\ t_2$	application
$\text{true}$	constant true
$\text{false}$	constant false
$\text{if } t_1 \text{ then } t_2 \text{ else } t_3$	conditional

$T ::= \text{Bool}$
$T_1 \rightarrow T_2$

# Formalização em Coq de STLC

```
Inductive ty : Type :=
| TBool : ty
| TArrow : ty -> ty -> ty.
```

```
Inductive tm : Type :=
| tvar : string -> tm
| tapp : tm -> tm -> tm
| tabs : string -> ty -> tm -> tm
| ttrue : tm
| tfalse : tm
| tif : tm -> tm -> tm -> tm.
```

## Semântica Small-Step

- A semântica *small-step* é uma relação de termo para termo que define como os passos atômicos da computação ocorrem.
- No Cálculo Lambda o passo da computação é a redução-beta.
- Redução-beta é definida por meio da regra de substituição.

## Regras de substituição

Utiliza-se uma versão simplificada das regras de substituição, onde  $s$  é sempre um termo fechado (sem variáveis livres).

$[x:=s]x$	$= s$	
$[x:=s]y$	$= y$	if $x \neq y$
$[x:=s](\lambda x:T_{11}. t_{12})$	$= \lambda x:T_{11}. t_{12}$	
$[x:=s](\lambda y:T_{11}. t_{12})$	$= \lambda y:T_{11}. [x:=s]t_{12}$	if $x \neq y$
$[x:=s](t_1 t_2)$	$= ([x:=s]t_1) ([x:=s]t_2)$	
$[x:=s]\text{true}$	$= \text{true}$	
$[x:=s]\text{false}$	$= \text{false}$	
$[x:=s](\text{if } t_1 \text{ then } t_2 \text{ else } t_3) =$		
	$\text{if } [x:=s]t_1 \text{ then } [x:=s]t_2 \text{ else } [x:=s]t_3$	

# Formalização em Coq das Regras de substituição como Fixpoint

Reserved Notation "'[ ' x ' :=' s ']' ' t" (at level 20).

```

Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
match t with
| tvar x'      => if beq_string x x' then s else t
| tabs x' T t1 => tabs x' T (if beq_string x x'
                                then t1 else ([x:=s] t1))
| tapp t1 t2   => tapp ([x:=s] t1) ([x:=s] t2)
| ttrue         => ttrue
| tfalse        => tfalse
| tif t1 t2 t3 => tif ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
end
where "'[ ' x ' :=' s ']' ' t" := (subst x s t).

```

## Regras *Small Step*

A computação no Cálculo Lambda é definida pela Redução Beta, a qual substitui o argumento pela variável ligada na abstração lambda. A Redução Beta, aqui, é denotada por  $(\lambda x:T.t12) v2 ==> [x:=v2]t12$ . Para a seguinte definição de *value*:

```

Inductive value : tm -> Prop :=
  v_abs : forall (x : string) (T : ty) (t : tm),
    value (tabs x T t)
  | v_true : value ttrue
  | v_false : value tfalse
  
```

as regras *Small Step* definem uma avaliação *eager* conforme:

$$\frac{\text{value } v2}{(\lambda x:T.t12) v2 ==> [x:=v2]t12} (\text{AppAbs})$$

## Regras *Small Step*

$$\frac{t1 \implies t1'}{t1\ t2 \implies t1'\ t2} (\text{App1})$$

$$\frac{\text{value } v1 \quad t2 \implies t2'}{v1\ t2 \implies v1\ t2'} (\text{App2})$$

Regras para condicionais:

$$\frac{}{( \text{if true then } t1 \text{ else } t2 ) \implies t1} (\text{IfTrue})$$

$$\frac{}{( \text{if false then } t1 \text{ else } t2 ) \implies t2} (\text{IfFalse})$$

$$\frac{t1 \implies t1'}{( \text{if } t1 \text{ then } t2 \text{ else } t3 ) \implies (\text{if } t1' \text{ then } t2 \text{ else } t3)} (\text{If})$$

# Formalização em Coq das Regras *Small Step*

```

Reserved Notation "t1 '==>' t2" (at level 40).
Inductive step : tm -> tm -> Prop :=
| ST_AppAbs : forall x T t12 v2,
  value v2 ->
  (tapp (tabs x T t12) v2) ==> [x:=v2]t12
| ST_App1 : forall t1 t1' t2,
  t1 ==> t1' ->
  tapp t1 t2 ==> tapp t1' t2
| ST_App2 : forall v1 t2 t2',
  value v1 -> t2 ==> t2' ->
  tapp v1 t2 ==> tapp v1 t2'
where "t1 '==>' t2" := (step t1 t2).

```

# Sistema de Tipos

- Interpreta-se a tripla relação  $\Gamma \vdash t : T$  como: Sobre as suposições  $\Gamma$ , o termo  $t$  tem o tipo  $T$ .
- O sistema de tipos é dado pelas regras de inferência abaixo:

$$\frac{}{\Gamma, x : T \vdash x : T} (\text{Var})$$

$$\frac{\Gamma, x : T_{11} \vdash t_{12} : T_{12}}{\Gamma \vdash (x:T_{11}.t_{12}) : T_{11} \rightarrow T_{12}} (\text{Abs})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} (\text{App})$$



# Sistema de Tipos

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} (\text{True})$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} (\text{False})$$

$$\frac{\Gamma \vdash t1 : \text{Bool} \quad \Gamma \vdash t2 : T \quad \Gamma \vdash t3 : T}{\Gamma \vdash \text{if } t1 \text{ the } t2 \text{ else } t3 : T} (\text{If})$$

# Formalização em Coq do Sistema de Tipos

```
Definition context := partial_map ty.
```

```
Reserved Notation "Gamma '|-' t '\in' T" (at level 40).
```

```
Inductive has_type : context -> tm -> ty -> Prop :=
```

```
| T_Var : forall Gamma x T, Gamma x = Some T ->
```

```
    Gamma |- tvar x \in T
```

```
| T_Abs : forall Gamma x T11 T12 t12,
```

```
    Gamma & {{x --> T11}} |- t12 \in T12 ->
```

```
    Gamma |- tabs x T11 t12 \in TArrow T11 T12
```

```
| T_App : forall T11 T12 Gamma t1 t2,
```

```
    Gamma |- t1 \in TArrow T11 T12 ->
```

```
    Gamma |- t2 \in T11 ->
```

```
    Gamma |- tapp t1 t2 \in T12
```

```
where "Gamma '|-' t '\in' T" := (has_type Gamma t T).
```

## Soundness

- A principal propriedade de um sistemas de tipos de uma linguagem de programação é *Soundness*.
- *Soundness* diz que todos os termos bem tipados nunca entram num estado *stuck*.
- Estado *stuck* é um termo que está na forma normal, mas não é um valor (valores aqui são true e false).
- *Soundness* pode ser representado por *Progress* + *Preservation*.

```
Definition stuck (t:tm) : Prop :=  

  (normal_form step) t /\ ~ value t.
```

```
Corollary soundness : forall t t' T,  

  empty |- t \in T -> t ==>* t' ->  

  ~(stuck t').
```

# Progress

A propriedade de progresso diz que todo termo fechado e bem tipado não está travado *stuck*, ou seja, ou é um valor ou pode fazer um passo de redução.

```
Theorem progress : forall t T,  
empty |- t \in T ->  
value t \vee exists t', t ==> t'.
```