

# Uma Certificação em Coq do Algoritmo W Monádico

Rafael Castro

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

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

22 de Agosto de 2019

## Introdução - Sistemas de Tipos

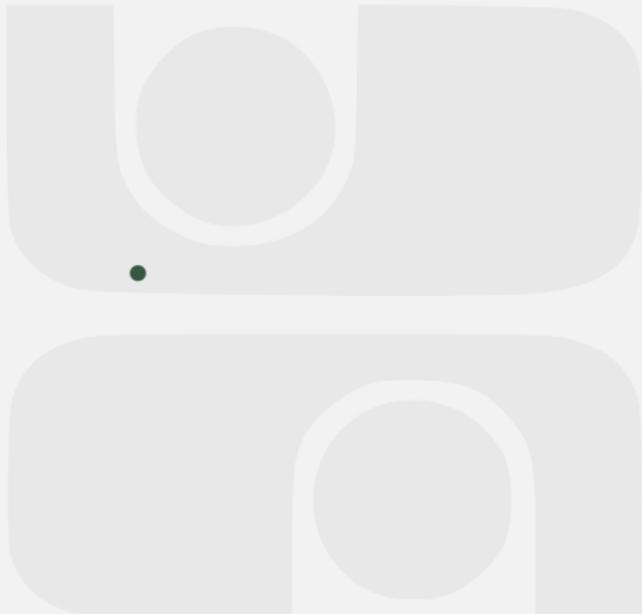
- Sistemas de tipos são relações que ditam quais expressões podem ser associadas a quais tipos.
- Nas linguagens de programação, classificavam os dados para serem tratados corretamente pelo processador;
- Passaram a ser utilizados como uma técnica para identificar falhas na consistência de programas
- Tipagem estática e dinâmica:
  - Estática: atribuições de tipo não mudam em *run time*; uma forma de verificação automática.
  - Dinâmica: atribuições de tipo **podem mudar** em *run time*; mais permissível; mais permissível a erros.  
`[1, 'a', objeto, [2]]`
- Propriedade de *soundness*:  
*Well-typed programs cannot go wrong.*

# Introdução - Inferência de Tipos e Polimorfismo

- A **inferência de tipos** é o processo de encontrar a assinatura de tipo de um programa.
- Liberdade ao programador de escolher anotar ou não os tipos das funções.
- Polimorfismo é quando uma função pode assumir vários usos com diferentes tipos de dados. Em especial, o polimorfismo paramétrico é feito pela quantificação de variáveis de tipos.

```
reverse :: [a] -> [a]
```

# Introdução - Algoritmo para Inferência de Tipos



# Fundamentos - Formalismos de Sistemas de Tipos

- Sistemas de tipos quando definidos formalmente podem garantir importantes propriedades da semântica de programas, como o famoso lema de Robin Milner “Well-typed programs cannot go wrong”.
- Damas-Milner: polimorfismo via *let*. Base para diversas linguagens de programação funcional. Não permite recursão polimórfica pela regra *fix*.
- O primeiro sistema de tipos a permitir recursão polimórfica é o Sistema-F.
- Milner-Mycroft altera a regra de definições recursivas *fix* para permitir recursões polimórficas.

$$\frac{\Gamma, \mathbf{x} : \tau \vdash e : \tau}{\Gamma \vdash \text{fix } x.e : \tau} (\text{fix}) \quad \frac{\Gamma, \mathbf{x} : \sigma \vdash e : \sigma}{\Gamma \vdash \text{fix } x.e : \sigma} (\text{fix+})$$

## Fundamentos - Inferência no Milner-Mycroft

- A inferência no sistema Milner-Mycroft é equivalente ao problema da semi-unificação.
- A semi-unificação é uma generalização da unificação.
- A semi-unificação é indecidível.
  - Casos decidíveis: apenas uma igualdade, qualquer aridade e no máximo uma variável livre, duas variáveis e semi-unificação acíclica.
- No caso geral não há como decidir a tipabilidade (*typability*) de um termo no sistema Milner-Mycroft.
- *Big type vs small type*.
- *Small types* no sistema Milner-Mycroft é decidível em tempo polinomial.
- Buscou-se alternativas para contornar a dificuldade deste problema.

# Fundamentos - Propostas de Recursão Polimórfica

- Somente existem semi-algoritmos e algoritmos totais, mas incompletos pois rejeitam alguns programas bem tipados.
- O semi-algoritmo para resolver a semi-unificação apresentado em (HENGLEIN, 1993) foi utilizado na construção do semi-algoritmo W+ em (EMMS; LEIL, 1999).
  - Damas-Milner tem pior caso exponencial. Não acontece em situações práticas. Prática vs teoria no Milner-Mycroft.
- Diversas propostas para recursão polimórfica foram apresentadas na literatura com o uso de *intersection types*.

## Fundamentos - Assistentes de provas

- São programas para o desenvolvimento de provas formais.
- O núcleo é um verificador, que verifica a consistência lógica da prova.
- Os principais apelos são:
  - a verificação mecânica é rápida e evita as falhas humanas;
  - interatividade, permite visualizar informações sobre os estados da prova;
  - comandos para busca de teoremas e lemas para o progresso da prova;
  - automatização de provas com métodos não-deterministas;
  - potencialização da capacidade humana de realizar provas;
  - extração de programas verificados.

Assistentes de provas permitem provar coisas que não seriam realizáveis somente com papel e caneta!

# Fundamentos - Por que Coq?

- Existem dezenas: Automath, Agda, Twelf, ACL2, PVS, Minlog, Isabelle e Coq...
- O núcleo do Coq é o *Calculus of Inductive Constructions* (CIC). Extensão do *Calculus of Constructions* (CoC).
- CoC é um Cálculo Lambda polimórfico de ordem superior e com tipos dependentes.
- **Justificativa:**
  - ① Frequent use of Coq for formalizing aspects of programming languages. CoqPL.
  - ② Relevant projects in Coq: CompCert is a compiler for C; Simplicity is a programming language for blockchains; Vellvm is a formalization of LLVM in Coq.
  - ③ Didactic materials available!

## Fundamentos - Terminação em Coq

- Coq é uma linguagem de programação funcional total.
- A garantia de terminação do Coq é uma conservadora verificação sintática: recursão primitiva.
- Coq conta com algumas maneiras de estender o conjunto de programas que podem ser implementados:
  - ① Recursão bem-fundada (*Well-Founded Recursion*) por meio de uma relação bem-fundada (*Well-Founded Relation*). Uma relação bem-fundada.
  - ② Recursão limitada (*Bounded Recursion*) através de um argumento, que representa o número de chamadas recursivas. Suficiente para que a computação finalize com o resultado correto.
  - ③ Recursão por iteração (*Recursion by Iteration*). Similar a recursão limitada. Definição de um funcional, que tem como um dos argumentos a função  $f$  que deseja-se implementar.

## Fundamentos - Terminação em Coq

- ④ Recursão sobre um predicado (*Recursion on an Ad Hoc Predicate Section*). Representa o domínio da função.  
Recursão estrutural sobre provas do princípio indutivo deste predicado.
- ⑤ Recursão por mònada inspirada em Teoria de Domínios.  
Combinadores de ordem superior representam funções que não terminam. Uma obrigação de prova é a continuidade da computação na mònada.
- ⑥ Recursão geral baseada numa mònada de um tipo co-indutivo.  
É similar ao anterior, mas utiliza os tipos co-indutivos para representar a noção de não terminação, com a vantagem de não ter obrigações de provas.
- ⑦ Linguagem embutida: formalizar completamente uma linguagem de programação.

## Fundamentos - Vantagens Terminação em Coq

- Somente as quatro primeiras técnicas permitem utilizar o mecanismo de avaliação interno do Coq.
- As técnicas 2 e 3 necessitam conhecer uma função capaz de computar o número de chamadas recursivas suficientes para uma dada entrada.
- A principal vantagem das últimas três técnicas é a possibilidade de implementar qualquer função recursiva e evitam provas.
- No aspecto da extração, a técnica mais eficiente é a primeira, pois as demais podem carregar (na extração) alguns lixos como os argumentos adicionais.

## MMo - Diferenças básicas

O MMo funciona de maneira *bottom-up* assim como o algoritmo *W*, ou seja, primeiro inferem-se os termos mais internos da expressão para seguir inferindo os mais externos.

- ① As variáveis com sequência de índices. Detecção de dependências circulares em substituições.
- ② Um contexto  $\Gamma$  é um conjunto de triplas (*variável*, *Kind\_def*, *scheme*). Não há condição de consistência. *Typing context*.
- ③ o MMo retorna uma dupla com o tipo simples inferido  $\tau$  e um contexto  $\Gamma$ .
- ④ Para cada definição num *BindGroup* infere-se o respectivo conjunto com os pares de identificadores e os seus respectivos tipos inferidos e requeridos (dado pelo símbolo  $\Sigma$ ).

## MMo - Inferência de grupo de definições

- ① Computar o conjunto  $\Sigma$  com todos os tipos inferidos e requeridos do grupo.
- ② Gera-se a partir de  $\Sigma$  um conjunto  $\Omega$  de pares tipos inferidos (instâncias) e requeridos ( $\tau', \tau$ ) que precisam ser unificados.
  - o tipo de cada suposição  $x:\tau$  em cada um dos  $\Gamma$  em  $\Sigma$  precisa ser unificado com uma instância do seu respectivo tipo inferido  $\tau'$  para  $x$ .
  - A função `supInst` troca cada variável de tipo  $\alpha^s$  de  $\tau'$  por uma nova instância  $\alpha^{i::s}$ .
- ③ Verificar a existência de dependências circulares na substituição  $\mathbb{S}$  gerada pela unificação de  $\Omega$ .
- ④ Caso exista dependência circular em  $\mathbb{S}$ , então o MMo termina com erro. Caso contrário, então aplica-se a  $\mathbb{S}$  em  $\Sigma$ . Se  $\mathbb{S}$  for efetiva, então volta-se para segunda etapa. Se não, para.

## MMo - Dependências circulares

- Uma dependência circular ocorre quando uma substituição troca uma variável de tipo  $\alpha^s$  por um tipo  $\tau$ , o qual ocorre uma variável de tipo  $\alpha^{s'}$  tal que  $s$  seja uma subsequência de  $s'$ . Ex:  $[a^{1,0} \rightarrow b/a^0]$ .
- A relação de ser subsequência pode ser indireta, portanto é necessário fazer uma verificação transitiva na substituição.

## MMo - Exemplo inferência de grupo

```

data Seq a = Nil | Cons a (Seq (a,a)).

len :: Seq a -> Int
len Nil = 0
len (Cons x s) = 1 + 2 * (len s)
  
```

$\Sigma$ :

```

{ (Nil, (Seq a, ∅)),
(Cons, (b → Seq (b, b) → Seq b, ∅)),
(len, (Seq c → Int,
      { (+): Int → Int → Int, len: Seq(c, c) → Int }))}  
```

$\Omega$ :

```
{ (Seq c0 → Int, Seq(c, c) → Int) }  
```

$\mathbb{S}$ :  $[(c, c)/c^0]$

Não tem dependência circular e não modifica os tipos de  $\Sigma$   
 (ignorando renomeações de variáveis)

## Formalização em Coq - Detalhes da formalização

- Formalização parcial e simplificada (sem grupos).
- Trabalho similar para o algoritmo  $W$  por (DUBOIS; MÉNISSIER-MORAIN, 1999).
- Uso da formalização da unificação feita por (RIBEIRO; CAMARÃO, 2016).
- Termos, tipos, *schemes* formalizados como tipos indutivos.
- Para garantir que dois tipos generalizados sejam sintaticamente iguais, a respeito da equivalência  $\alpha$ , utiliza-se a notação *De Bruijn* nos *schemes*.
- Uso de mônadas de estados e falhas.

## Formalização em Coq - Aspectos da terminação

- Coq é uma linguagem de programação funcional total.
- De todas as funções do MMo, a que computa o conjunto  $\Sigma$  é a única que não é de recursão primitiva.
- Até o momento não se conhece alguma relação bem fundada a respeito da efetividade da substituição.
- Utilizar alguma forma de limite de iteração.
- Trabalho similar para o algoritmo  $W$  por (DUBOIS; MÉNISSIER-MORAIN, 1999).
- Uso da formalização da unificação feita por (RIBEIRO; CAMARÃO, 2016).
  - Seria necessário descobrir como computar esse número ou, então, associar esse valor às regras de inferência.

## Formalização em Coq - Aspectos da consistência

- Uso das regras de Damas-Milner em *syntax-directed* em DUBOIS; MÉNISSIER-MORAIN, 1999).
  - Regras da forma  $\Gamma \vdash e : \tau$ .
  - Regras de especialização (*spec*) e generalização (*gen*) são fundidas, respectivamente, com as regras (*var*) e (*let*).
- Regra (*fix+*) na forma *syntax-directed*.
- As regras de inferência são representadas em Coq como um tipo (proposição) indutivo.
- O tipo indutivo é uma certificação que pode ser diretamente utilizada num tipo dependente no MMo:

```
Definition infer_dep : forall (e : term) (G : ctx),  
  Infer ({tau : ty &  
    {g : ctx | has_type (apply_subst_ctx s G) e tau}})
```

# Formalização em Coq - Estabilidade da substituição

- O lema da estabilidade da substituição é uma propriedade clássica em sistemas de tipos e é fundamental na prova da consistência.
- Se é verdade que  $\Gamma \vdash e : \tau$ , então para qualquer substituição  $\mathbb{S}$  tem-se  $\mathbb{S}\Gamma \vdash e : \mathbb{S}\tau$ .
- Casos monomórficos são fáceis. Além do caso polimórfico do `let_ht` há também o caso do `fix_ht`.

# Conclusão

- Fundamentos teóricos para formalização do MMo.
- Assistentes de provas Coq.
- Como o MMo funciona.
- Formalização do MMo parcial.
- Principais dificuldades encontradas.
- Alternativas para a prova de terminação.