

Um Algoritmo Verificado para Inferência de Tipos na Presença de Recursão Polimórfica

Rafael Castro

rafaelcgs10@gmail.com

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

15 de Fevereiro de 2019

Introdução - Sistemas de Tipos

- Sistemas de tipos são regras 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 regras 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 **mudam** em *run time*; mais permissível; mais permissível a erros.

[1, 'a', objeto, [2]]

Introdução - Inferência de Tipos e Polimorfismo

- A inferência de tipos é o processo de encontrar a assinatura de tipo mais geral 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.

`len :: [a] -> Int`

- Existem situações onde a inferência de tipos é mais difícil que a verificação. Nesses casos, algoritmos podem rejeitar programas bem tipados.

Introdução - Recursão Polimórfica

- A recursão polimórfica acontece quando o tipo da chamada recursiva de uma função pode mudar ao longo das recursões.

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

```
len :: Seq a -> Int
```

```
len Nil = 0
```

```
len (Cons x s) = 1 + 2 * (len s)
```

- Haskell rejeita programas com recursão polimórfica se a assinatura de tipo não for explicitamente anotada.
- Diversos exemplos de exemplos precisando de recursão polimórfica foram apresentados na literatura.

Introdução - Motivação e Objetivo

- O semi-algoritmo MMo proposto por (VASCONCELLOS; CAMARÃO, 2003) consegue tipar diversos casos de recursão polimórfica.
- MMo não tem as suas propriedades fundamentais provadas: terminação (totalidade) e consistência (correto).
- **Objetivo:** investigar se o semi-algoritmo MMo tem as propriedades de totalidade e correção em relação ao seu sistema de tipos (especificação).
- A especificação em si também é um elemento de investigação.
- O método utilizado é a formalização do MMo e a sua especificação no assistente de provas Coq.

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.

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

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.
- Se o tamanho de um tipo não é razoável ao do seu programa, é exponencial por exemplo, então esse programa é dito ter um *big type* ou um *small type* caso contrário.
- *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).
 - Argumenta-se que assim como a inferência em Damas-Milner tem pior caso exponencial e que não acontece em situações práticas, uma diferença similar entre prática e teoria acontece na inferência do Milner-Mycroft.
- Diversas propostas para recursão polimórfica foram apresentadas na literatura com o uso de *intersection types*.

Fundamentos - 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.
- Os principais apelos para o uso de assistentes de provas 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 de assistentes de provas: Automath, Agda, Twelf, ACL2, PVS, Minlog, Isabelle e Coq...
- O assistente de provas Coq tem como *kernel* (núcleo) o formalismo *Calculus of Inductive Constructions* (CIC), uma extensão do Cálculo Lambda Tipado conhecido como *Calculus of Constructions* (CoC).
- CoC é classificado como um Cálculo Lambda polimórfico de ordem superior e com tipos dependentes.
- **Justificativa:**
 - ① Frequent use of Coq for formalizing aspects of programming languages. CoqPL: a track dedicated to the use of Coq at POPL.
 - ② Relevant projects in Coq: CompCert is a C compiler; 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*). Os argumentos da chamada recursivas obedecem uma relação bem-fundada.
 - ② Recursão limitada (*Bounded Recursion*) através de um argumento, que representa o número de chamadas recursivas. Esse argumento contador precisa ser computado previamente e deve ser suficiente para que a computação finalize com o resultado correto.
 - ③ Recursão por iteração (*Recursion by Iteration*) é uma técnica similar a recursão limitada, mas começa com a definição de um funcional uma definição que tem como um dos argumentos a função f que deseja-se implementar.

Fundamentos - Terminação em Coq

- ① Recursão sobre um predicado (*Recursion on an Ad Hoc Predicate Section*) definido por um tipo indutivo, que serve para representar o domínio da função e fazer recursão estrutural sobre provas do princípio indutivo deste predicado.
- ② Recursão por mònada inspirada em Teoria de Domínios, cujos combinadores de ordem superior servem para representar até mesmo funções que não terminam. Uma obrigação de prova é a continuidade da computação na mònada.
- ③ 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.
- ④ Linguagem embutida: formalizar completamente uma linguagem de programação com recursão geral por meio dos tipos indutivos de Coq e, então, implementar os programas nesta linguagem embutida.