Tipos

- A verificação de tipos é uma operação mais sutil, onde está o erro nesta expressão?

```
(+ 3 (func x (x)))
```

Se tentarmos executar, o erro aparece no nível sintático, mas na verdade é um erro *semântico*.
Não é um erro de sintaxe!
Este caso é trivial perceber o erro, dá para corrigir até no *parser*.

Tipos

- A verificação de tipos é uma operação mais sutil, onde está o erro nesta expressão?

```
  (+ 3 (func x (x)))
```

  Se tentarmos executar, o erro aparece no nível sintático, mas na verdade é um erro *semântico*.
  Não é um erro de sintaxe!
  Este caso é trivial perceber o erro, dá para corrigir até no *parser*.

- Este outro também é fácil de verificar:

```
(:= f (func  x (+ x 1))
  (+ 3 (f 5)))
```

  `f` retorna um número, então ok

Casos mais chatos

Qual o problema aqui?

```
(:= f (func x  (func y (+ x y))))
      (+ 3 (f 5)))
```

E neste caso?

```
(func f (+ 3 (f 5)))
```

Casos mais chatos

Qual o problema aqui? `f` retorna uma função!

```
(:= f (func x  (func y (+ x y))))
      (+ 3 (f 5)))
```

E neste caso?

```
(func f (+ 3 (f 5)))
```

Casos mais chatos

Qual o problema aqui? `f` retorna uma função!

```
(:= f (func x  (func y (+ x y))))
      (+ 3 (f 5)))
```

E neste caso? Será que `f 5` retorna um número?

```
(func f (+ 3 (f 5)))
```

Casos mais chatos

Qual o problema aqui? `f` retorna uma função!

```
(:= f (func x  (func y (+ x y))))
      (+ 3 (f 5)))
```

E neste caso? Será que `f 5` retorna um número?

```
(func f (+ 3 (f 5)))
```

Como saber se é válido ou não sem que o programa seja executado?

Quando?

Quando fazer a verificação de tipo? Na compilação ou na execução?

```
(+ 3 (if0 algo 5 (fun (x) x)))
```

ou

```
(+ 3 (if0 (read-number) 5 (func x x)))
```

ou ainda a conjectura de Collatz, etc

Verificação de Tipos

Verificar tipos está intimamente ligada ao *Problema da Parada*. Não temos informação suficiente em tempo de compilação para determinar se os tipos são consistentes ou não. Os sistemas de tipo fazem verificação aproximada.

- Podem ser aceitos programas com erros de execução
- Podem rejeitar programas que podem funcionar

É um problema longe de ser resolvido. O uso de *cast* é uma "gambiarra"

Tipo

Tipo é qualquer propriedade de um programa que pode ser verificada sem executá-lo. Normalmente se considera tipo como um conjunto de valores possíveis.

Nem sempre é possível verificar tipos.

Cada expressão pode ser associada a um tipo, e tipos devem ser consistentes, mas nem todos os erros são erros de tipo.

Problemas

Incluir mais tipos produz menos erros, mas traz problemas:

- Aumentam as restrições, talvez a um ponto inaceitável
- Podem aumentar o custo computacional
- Exige anotação no programa
- Pode atingir o limite da computabilidade

Problemas

Incluir mais tipos produz menos erros, mas traz problemas:

- Aumentam as restrições, talvez a um ponto inaceitável
- Podem aumentar o custo computacional
- Exige anotação no programa
- Pode atingir o limite da computabilidade

Será que vale a pena?

- Evita tempo de depuração
- Captura erros em trechos ainda não executados
- Documentação
- Ajudam o compilador
- Forçam código mais elegante

Sistemas de tipos

Um sistema de tipos é composto por 3 partes:

- Uma coleção de tipos
- Um julgamento para definir se um valor corresponde a um dado tipo
- Um algoritmo para exercer este julgamento

É preciso definir regras para cada expressão.

```
[n : Número]
[(func a b) : Função]
```

O mapeamento é definido por um *ambiente de tipos* (*type environment*)

Ambiente de tipos e julgadores

O ambiente de tipos é designado por Γ
Um julgador de tipos é indicado assim:

$$\Gamma \vdash e : t$$

indicando que a expressão e tem tipo t, dizemos que Γ demonstra este fato.

$$\Gamma \vdash n : \text{Número}$$

$$\Gamma \vdash (\texttt{func}\ \ a\ \ b) : \text{Função}$$

$$\Gamma \vdash i : \Gamma(i)$$

Julgamento de tipos

Podemos escrever regras, da mesma forma que foi feito com a semântica.

$$\frac{\Gamma \vdash \mathsf{l} : \mathit{num} \qquad \Gamma \vdash \mathsf{r} : \mathit{num}}{\Gamma \vdash \texttt{(+ l r)} : \mathit{num}}$$

Aplicação de funções?

$$\frac{\Gamma \vdash \mathsf{f} : \mathit{func} \qquad \Gamma \vdash \mathsf{a} : \tau_a}{\Gamma \vdash \texttt{(f a)} : ???}$$

É preciso incluir anotação.

Anotação

```
(func (n : num) : num (+ x 3))
```

Os tipos ficam definidos e agora é possível, desde que o programador não minta, inferir o resultado.

$$\frac{\Gamma \vdash \mathsf{f} : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash \mathsf{a} : \tau_1}{\Gamma \vdash \texttt{(f a)} : \tau_2}$$

A linguagem muda também:

```
Expr: ...
     | (func (<id>:<tipo>):<tipo> <Expr> )

<tipo> : num
       | (<tipo>-><tipo>)
```

Completo

Completando, para garantir consistência:

$$\frac{\Gamma[i \leftarrow \tau_1] \vdash b \rightarrow \tau_2}{\Gamma \vdash (\texttt{func}\ \ i\ \ b) : \tau_1 \rightarrow \tau_2}$$

Na aplicação, forçamos o tipo correto e esperamos que o resultado esteja certo.

Teste

```
(+ 2 (+ 3 4))
```

$$\dfrac{\Phi \vdash 2 : num \qquad \dfrac{\Phi \vdash 3 : num \quad \Phi \vdash 4 : num}{\Phi \vdash \texttt{(+ 3 4)} : num}}{\Phi \vdash \{+ \ 2 \ \{ \ + \ 3 \ 4\}\} : num}$$

Com função

```
((func (x : number) : number
   (+ x 3))
 5)

(+ 3
   (func (x : number) : number
     x))
```

Falha

não conseguimos montar a árvore de tipos, daí a indicação de erro.

Não é a arvore em si que detecta.

Recursão

```
(rec (<id>:<tipo> Expr) Expr)
```

Recursão

```
(rec (<id>:<tipo> Expr) Expr)
```

$$\frac{?????????}{\Gamma \vdash \texttt{(rec (i:}\tau_a\ \nu\texttt{):b)} : \tau}$$

Recursão

```
(rec (<id>:<tipo> Expr) Expr)
```

$$\frac{\Gamma[i \leftarrow \tau_a] \vdash b : \tau \quad \Gamma[i \leftarrow \tau_a] \vdash \nu : ??}{\Gamma \vdash \texttt{(rec (i:}\tau_a \ \nu\texttt{):b)} : \tau}$$

Recursão

```
(rec (<id>:<tipo> Expr) Expr)
```

$$\frac{\Gamma[i \leftarrow \tau_a] \vdash b : \tau \quad \Gamma[i \leftarrow \tau_a] \vdash \nu : \tau_a}{\Gamma \vdash \texttt{(rec (i:}\tau_a\ \nu\texttt{):b)} : \tau}$$