

Functional Programming

Evaluation and Typing

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

Results of computations

Normal forms

- expensive to compute
- rarely needed for evaluation
- key to expense: evaluation under lambda

Results of computations

Normal forms

- expensive to compute
- rarely needed for evaluation
- key to expense: evaluation under lambda

Definition: Weak head-normal form

A pure lambda term is in **weak head-normal form** (or a **value**) iff it has the form

$$V ::= \lambda x.M$$

All other terms are **non-values**.

Deterministic Evaluation

Definition

A **reduction strategy** is a function that assigns to each closed, non-value lambda term the position of the next reduction.

Deterministic Evaluation

Definition

A **reduction strategy** is a function that assigns to each closed, non-value lambda term the position of the next reduction.

Reduction strategy: Call-by-name (related to Haskell)

$$(\lambda x.M) N \rightarrow_{\beta} M[x \mapsto N] \qquad \frac{M \rightarrow_{\beta} M'}{(M N) \rightarrow_{\beta} (M' N)}$$

Deterministic Evaluation

Definition

A **reduction strategy** is a function that assigns to each closed, non-value lambda term the position of the next reduction.

Reduction strategy: Call-by-name (related to Haskell)

$$(\lambda x.M) N \rightarrow_{\beta} M[x \mapsto N] \qquad \frac{M \rightarrow_{\beta} M'}{(M N) \rightarrow_{\beta} (M' N)}$$

Reduction strategy: Call-by-value (based on β -value reduction)

$$(\lambda x.M) \textcolor{red}{V} \rightarrow_{\beta V} M[x \mapsto \textcolor{red}{V}] \qquad \frac{M \rightarrow_{\beta V} M'}{(M N) \rightarrow_{\beta V} (M' N)} \qquad \frac{N \rightarrow_{\beta V} N'}{(\textcolor{red}{V} N) \rightarrow_{\beta V} (\textcolor{red}{V} N')}$$

Evaluation in Functional Programming Languages

Evaluation of closed terms

- Reduction is restricted to **closed terms** and stops at weak head-normal forms.

Evaluation in Functional Programming Languages

Evaluation of closed terms

- Reduction is restricted to **closed terms** and stops at weak head-normal forms.
- Consequence: substitution need not rename variables!

Evaluation in Functional Programming Languages

Evaluation of closed terms

- Reduction is restricted to **closed terms** and stops at weak head-normal forms.
- Consequence: substitution need not rename variables!
- Substitution can be avoided by implementation tricks

Evaluation in Functional Programming Languages

Evaluation of closed terms

- Reduction is restricted to **closed terms** and stops at weak head-normal forms.
- Consequence: substitution need not rename variables!
- Substitution can be avoided by implementation tricks
- Datatypes are not encoded, but added to the calculus

Applied Lambda Calculus

Syntax

Add constants as values to the pure lambda calculus

$$L, M, N ::= x \mid \lambda x.M \mid M N \mid C$$

$$C ::= \text{TRUE} \mid \text{FALSE} \mid \text{IF} \mid 0 \mid 1 \mid \dots \mid \text{SUCC} \mid \dots \mid \text{PAIR} \mid \text{FST} \mid \text{SND}$$

$$V, W ::= \lambda x.M \mid C \mid \text{IF } V \mid \text{IF } V M \mid \text{PAIR } M \mid \text{PAIR } M N$$

Semantics (call-by-name)

β reduction and δ **reduction rules** for the constants

$$(\lambda x.M) N \rightarrow_{\beta} M[x \mapsto N]$$

$$\text{IF TRUE } M N \rightarrow_{\delta} M$$

$$\text{IF FALSE } M N \rightarrow_{\delta} N$$

$$\text{FST}(\text{PAIR } M N) \rightarrow_{\delta} M$$

$$\text{SND}(\text{PAIR } M N) \rightarrow_{\delta} N$$

Handling Constants

Context rule for constants

$$\frac{N \rightarrow_x N' \quad V \in \{IF, FST, SND, SUCC\}}{(V \ N) \rightarrow_x (V \ N')}$$

Handling Constants

Context rule for constants

$$\frac{N \rightarrow_x N' \quad V \in \{IF, FST, SND, SUCC\}}{(V N) \rightarrow_x (V N')}$$

New source of errors: stuck terms

- Pure lambda calculus: closed term is either value or can reduce
- Applied lambda calculus: there are closed non-value terms that cannot be reduced
 - ▶ $TRUE V$
 - ▶ $IF(\lambda x.L) M N$
 - ▶ $FST(\lambda x.M)$
 - ▶ $IF(PAIR M N) M' N'$
 - ▶ ...
- These terms are **stuck terms**

Typing

Typing rules out stuck terms

What is a typing?

- Typing $M : T$ is a relation between terms and types
- Typing characterizes terms with a certain behavior

Typing rules out stuck terms

What is a typing?

- Typing $M : T$ is a relation between terms and types
- Typing characterizes terms with a certain behavior

Simple types for the applied lambda calculus

$$T ::= T \rightarrow T \mid Nat \mid Bool \mid Pair\ T\ T$$

Intended Behavior

If $M : T$, then M is not stuck.

Simple Types

Definition: Typing assumption (environment)

$A ::= \cdot \mid A, x : T$ if $x \notin A$ specifies typing assumption for variables

Simple Types

Definition: Typing assumption (environment)

$A ::= \cdot \mid A, x : T$ if $x \notin A$ specifies typing assumption for variables

Definition: Typing judgment

$A \vdash M : T$ “under typing assumption A , term M has type T ”

Simple Types

Definition: Typing assumption (environment)

$A ::= \cdot \mid A, x : T$ if $x \notin A$ specifies typing assumption for variables

Definition: Typing judgment

$A \vdash M : T$ “under typing assumption A , term M has type T ”

Definition: Typing rules — lambda calculus with numbers

VAR
 $A, x : T, A' \vdash x : T$

LAM
$$\frac{A, x : T \vdash M : T'}{A \vdash \lambda x. M : T \rightarrow T'}$$

APP
$$\frac{A \vdash M : T \rightarrow T' \quad A \vdash N : T}{A \vdash M N : T'}$$

NUM
 $A \vdash n : \text{Nat}$

SUCC
$$\frac{A \vdash M : \text{Nat}}{A \vdash \text{SUCC } M : \text{Nat}}$$

More typing rules

Definition: Typing rules — boolean fragment

TRUE

$A \vdash \text{TRUE} : \text{Bool}$

FALSE

$A \vdash \text{FALSE} : \text{Bool}$

IF

$A \vdash L : \text{Bool}$

$A \vdash M : T$

$A \vdash N : T$

$A \vdash \text{IF } L \text{ M } N : T$

More typing rules

Definition: Typing rules — boolean fragment

TRUE
 $A \vdash \text{TRUE} : \text{Bool}$

FALSE
 $A \vdash \text{FALSE} : \text{Bool}$

IF
$$\frac{A \vdash L : \text{Bool} \quad A \vdash M : T \quad A \vdash N : T}{A \vdash \text{IF } L \text{ } M \text{ } N : T}$$

Definition: Typing rules — pairs

PAIR
$$\frac{A \vdash M : T \quad A \vdash N : T'}{A \vdash \text{PAIR } M \text{ } N : \text{Pair } T \text{ } T'}$$

FST
$$\frac{A \vdash M : \text{Pair } T \text{ } T'}{A \vdash \text{FST } M : T}$$

SND
$$\frac{A \vdash M : \text{Pair } T \text{ } T'}{A \vdash \text{SND } M : T'}$$

Example Inference Tree

$$\frac{\frac{\dots \vdash f : \alpha \rightarrow \alpha \quad \frac{\dots \vdash f : \alpha \rightarrow \alpha \quad \dots \vdash x : \alpha}{\dots \vdash f x : \alpha}}{f : \alpha \rightarrow \alpha, x : \alpha \vdash f (f x) : \alpha}}{f : \alpha \rightarrow \alpha \vdash \lambda x. f (f x) : \alpha \rightarrow \alpha}}{\cdot \vdash \lambda f. \lambda x. f (f x) : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha}$$

Type Soundness

Type Preservation

If $\cdot \vdash M : T$ and $M \rightarrow N$, then $\cdot \vdash N : T$.

Proof by induction on $M \rightarrow N$.

Progress

If $\cdot \vdash M : T$, then either M is a value or there exists M' such that $M \rightarrow M'$.

Proof by induction on $A \vdash M : T$.

Type Soundness

If $\cdot \vdash M : T$, then either

- 1 exists V such that $M \rightarrow^* V$ or
- 2 for each N , such that $M \rightarrow^* N$ there exists N' such that $N \rightarrow N'$.

Type Inference for the Simply-Typed Lambda Calculus

Type Inference for the Simply-Typed Lambda Calculus (STLC)

Typing Problems

- Type checking: Given environment A , a term M and a type T , is $A \vdash M : T$ derivable?
- Type inference: Given a term M , are there A and T such that $A \vdash M : T$ is derivable?

Type Inference for the Simply-Typed Lambda Calculus (STLC)

Typing Problems

- Type checking: Given environment A , a term M and a type T , is $A \vdash M : T$ derivable?
- Type inference: Given a term M , are there A and T such that $A \vdash M : T$ is derivable?

Typing Problems for STLC

- Type checking and type inference are decidable for STLC
- Moreover, for each typable M there is a **principal typing** $A \vdash M : T$ such that any other typing is a substitution instance of the principal typing.

Prerequisites for Type Inference for STLC

Unification

Let \mathcal{E} be a set of equations on types.

Unifiers and Most General Unifiers

- A substitution S is a **unifier of \mathcal{E}** if, for each $T \doteq T' \in \mathcal{E}$, it holds that $ST = ST'$.
- A substitution S is a **most general unifier of \mathcal{E}** if S is a unifier of \mathcal{E} and for every other unifier S' of \mathcal{E} , there is a substitution U such that $S' = U \circ S$.

Prerequisites for Type Inference for STLC

Unification

Let \mathcal{E} be a set of equations on types.

Unifiers and Most General Unifiers

- A substitution S is a **unifier of \mathcal{E}** if, for each $T \doteq T' \in \mathcal{E}$, it holds that $ST = ST'$.
- A substitution S is a **most general unifier of \mathcal{E}** if S is a unifier of \mathcal{E} and for every other unifier S' of \mathcal{E} , there is a substitution U such that $S' = U \circ S$.

Unification

There is an algorithm \mathcal{U} that, on input \mathcal{E} , either returns a most general unifier of \mathcal{E} or fails if none exists.

Principal Type Inference for STLC

The algorithm (due to John Mitchell) transforms a term into a principal typing judgment for the term or fails if no typing exists.

```
 $\mathcal{P}(x)$  = return  $[x : \alpha \vdash x : \alpha]$   
 $\mathcal{P}(\lambda x.M)$  = let  $[A \vdash M : T] \leftarrow \mathcal{P}(M)$  in  
          if  $A$  has form  $A', x : T_x$  then return  $[A' \vdash \lambda x.M : T_x \rightarrow T]$   
          else choose  $\alpha \notin \text{var}(A, T)$  in  
              return  $[A \vdash \lambda x.M : \alpha \rightarrow T]$   
 $\mathcal{P}(M_0 M_1)$  = let  $[A_0 \vdash M_0 : T_0] \leftarrow \mathcal{P}(M_0)$  in  
          let  $[A_1 \vdash M_1 : T_1] \leftarrow \mathcal{P}(M_1)$  in  
          with disjoint type variables in  $(A_0, T_0)$  and  $(A_1, T_1)$   
          choose  $\alpha \notin \text{var}(A_0, A_1, T_0, T_1)$  in  
          let  $S \leftarrow \mathcal{U}(A_0 \dot{=} A_1, T_0 \dot{=} T_1 \rightarrow \alpha)$  in  
          return  $[SA_0 \cup SA_1 \vdash M_0 M_1 : S\alpha]$   
 $\mathcal{P}(n)$  = return  $[\cdot \vdash n : \text{Nat}]$   
 $\mathcal{P}(\text{SUCC } M)$  = let  $[A \vdash M : T] \leftarrow \mathcal{P}(M)$  in  
          let  $S \leftarrow \mathcal{U}(T \dot{=} \text{Nat})$  in  
          return  $[SA \vdash \text{SUCC } M : \text{Nat}]$ 
```

Properties of Type Inference

Soundness

If $\mathcal{P}(M) = [A \vdash M : T]$, then $A \vdash M : T$ is derivable.

Completeness

If $A \vdash M : T$ is derivable, then $\mathcal{P}(M)$ succeeds with result $[A' \vdash M : T']$ such that $A = SA'$ and $T = ST'$ for some substitution S .

Wrapup

- Call-by-name and call-by-value are deterministic evaluation strategies that are more efficient than full β reduction
- Applied lambda calculus contains constants that encode operations on datatypes
- Applied lambda calculus can have stuck terms
- Simple types avoid stuck terms
- Type checking and type inference for simple types is decidable
- There is a sound and complete algorithm for type inference for simple types