# Functional Programming

## Evaluation and Typing

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

# 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 $\beta$-value reduction)

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

# 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

$$
\begin{aligned}
L, M, N &::= x \mid \lambda x.M \mid M\,N \mid C \\
C &::= \mathit{TRUE} \mid \mathit{FALSE} \mid \mathit{IF} \mid 0 \mid 1 \mid \cdots \mid \mathit{SUCC} \mid \cdots \mid \mathit{PAIR} \mid \mathit{FST} \mid \mathit{SND} \\
V, W &::= \lambda x.M \mid C \mid \mathit{IF}\,V \mid \mathit{IF}\,V\,M \mid \mathit{PAIR}\,M \mid \mathit{PAIR}\,M\,N
\end{aligned}
$$

## Semantics (call-by-name)

$\beta$ reduction and $\delta$ **reduction rules** for the constants

$$
(\lambda x.M)\,N \rightarrow_\beta M[x \mapsto N] \qquad \mathit{IF}\,\mathit{TRUE}\,M\,N \rightarrow_\delta M \qquad \mathit{IF}\,\mathit{FALSE}\,M\,N \rightarrow_\delta N
$$

$$
\mathit{FST}\,(\mathit{PAIR}\,M\,N) \rightarrow_\delta M \qquad \mathit{SND}\,(\mathit{PAIR}\,M\,N) \rightarrow_\delta N
$$

# Handling Constants

## Context rule for constants

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

# Handling Constants

## Context rule for constants

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

## 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

$$T ::= T \to T \mid \mathit{Nat} \mid \mathit{Bool} \mid \mathit{Pair}\, T\ T$$

### Intended Behavior

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

# Simple Types

**Definition: Typing assumption (environment)**

$A ::= \cdot \mid A, x : T \text{ if } x \notin A$ 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)**

$A ::= \cdot \mid A, x : T \text{ if } x \notin A$ 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**

$$\textsc{Var}\quad A, x : T, A' \vdash x : T$$

$$\textsc{Lam}\quad \frac{A, x : T \vdash M : T'}{A \vdash \lambda x.M : T \to T'}$$

$$\textsc{App}\quad \frac{A \vdash M : T \to T' \qquad A \vdash N : T}{A \vdash M\,N : T'}$$

$$\textsc{Num}\quad A \vdash n : Nat$$

$$\textsc{Succ}\quad \frac{A \vdash M : Nat}{A \vdash SUCC\,M : Nat}$$

# More typing rules

**Definition: Typing rules — boolean fragment**

$$\textsc{True}\quad \frac{}{A \vdash \mathit{TRUE} : \mathit{Bool}}$$

$$\textsc{False}\quad \frac{}{A \vdash \mathit{FALSE} : \mathit{Bool}}$$

$$\textsc{If}\quad \frac{A \vdash L : \mathit{Bool} \qquad A \vdash M : T \qquad A \vdash N : T}{A \vdash \mathit{IF}\, L\, M\, N : T}$$

# More typing rules

**Definition: Typing rules — boolean fragment**

$$\textsc{True}\quad \dfrac{}{A \vdash \mathit{TRUE} : \mathit{Bool}} \qquad \textsc{False}\quad \dfrac{}{A \vdash \mathit{FALSE} : \mathit{Bool}} \qquad \textsc{If}\quad \dfrac{A \vdash L : \mathit{Bool} \qquad A \vdash M : T \qquad A \vdash N : T}{A \vdash \mathit{IF}\, L\, M\, N : T}$$

**Definition: Typing rules — pairs**

$$\textsc{Pair}\quad \dfrac{A \vdash M : T \qquad A \vdash N : T'}{A \vdash \mathit{PAIR}\, M\, N : \mathit{Pair}\, T\, T'} \qquad \textsc{Fst}\quad \dfrac{A \vdash M : \mathit{Pair}\, T\, T'}{A \vdash \mathit{FST}\, M : T} \qquad \textsc{Snd}\quad \dfrac{A \vdash M : \mathit{Pair}\, T\, T'}{A \vdash \mathit{SND}\, M : T'}$$

# Example Inference Tree

$$
\dfrac{\cdots \vdash f : \alpha \to \alpha \qquad \dfrac{\cdots \vdash f : \alpha \to \alpha \qquad \cdots \vdash x : \alpha}{\cdots \vdash f\,x : \alpha}}{\dfrac{f : \alpha \to \alpha, x : \alpha \vdash f\,(f\,x) : \alpha}{\dfrac{f : \alpha \to \alpha \vdash \lambda x.f\,(f\,x) : \alpha \to \alpha}{\cdot \vdash \lambda f.\lambda x.f\,(f\,x) : (\alpha \to \alpha) \to \alpha \to \alpha}}}
$$

# 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.

$$
\begin{array}{lcl}
\mathcal{P}(x) & = & \textbf{return } \lceil x : \alpha \vdash x : \alpha \rceil \\
\mathcal{P}(\lambda x.M) & = & \textbf{let } \lceil A \vdash M : T \rceil \leftarrow \mathcal{P}(M) \textbf{ in} \\
 & & \textbf{if } A \textbf{ has form } A', x : T_x \textbf{ then return } \lceil A' \vdash \lambda x.M : T_x \to T \rceil \\
 & & \textbf{else choose } \alpha \notin \mathrm{var}\,(A, T) \textbf{ in} \\
 & & \qquad \textbf{return } \lceil A \vdash \lambda x.M : \alpha \to T \rceil \\
\mathcal{P}(M_0\, M_1) & = & \textbf{let } \lceil A_0 \vdash M_0 : T_0 \rceil \leftarrow \mathcal{P}(M_0) \textbf{ in} \\
 & & \textbf{let } \lceil A_1 \vdash M_1 : T_1 \rceil \leftarrow \mathcal{P}(M_1) \textbf{ in} \\
 & & \textbf{with disjoint type variables in } (A_0, T_0) \textbf{ and } (A_1, T_1) \\
 & & \textbf{choose } \alpha \notin \mathrm{var}\,(A_0, A_1, T_0, T_1) \textbf{ in} \\
 & & \textbf{let } S \leftarrow \mathcal{U}(A_0 \doteq A_1, T_0 \doteq T_1 \to \alpha) \textbf{ in} \\
 & & \textbf{return } \lceil SA_0 \cup SA_1 \vdash M_0\, M_1 : S\alpha \rceil \\
\mathcal{P}(n) & = & \textbf{return } \lceil \cdot \vdash n : Nat \rceil \\
\mathcal{P}(SUCC\, M) & = & \textbf{let } \lceil A \vdash M : T \rceil \leftarrow \mathcal{P}(M) \textbf{ in} \\
 & & \textbf{let } S \leftarrow \mathcal{U}(T \doteq Nat) \textbf{ in} \\
 & & \textbf{return } \lceil SA \vdash SUCC\, M : Nat \rceil
\end{array}
$$

# Properties of Type Inference

## Soundness

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

## Completeness

If $A \vdash M : T$ is derivable, then $\mathcal{P}(M)$ succeeds with result $\lceil A' \vdash M : T' \rceil$ 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 $\beta$ 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